Indice

1 Introduzione

Lo scopo di questo documento è quello di fornire un riferimento a livello introduttivo a quanti vorranno sviluppare software per Macintosh.
Il testo è stato pensato per coloro che, pur avendo già una certa esperienza nello sviluppo di programmi, non hanno ancora avuto esperienze di sviluppo su Mac, e per coloro che ancora non si sono familiarizzati con l'uso del vasto insieme di routines di sistema di Mac.

Macintosh è una macchina molto differente dalla maggior parte dei personal computer oggi sul mercato, sia per quanto riguarda il tipo di interazione con l'utente finale, sia per quanto riguarda il processo di ideazione e sviluppo di programmi. Tutti i programmi ruotano attorno ad un vasto insieme di routines predefinite e residenti in ROM (il Toolbox) che garantiscono una gestione uniforme e integrata di strumenti quali i menù pull-down, le finestre, il mouse, i controls, ecc., che rendono Mac unico.

Proprio la vastità e complessità del Toolbox però crea spesso dei problemi al primo approccio di un programmatore non ancora abituato all'ambiente Mac; egli si trova spaesato di fronte a circa 500 routines e ad un manuale di oltre 1000 pagine come *Inside Macintosh*, che, pur essendo estremamente accurato e dettagliato, non porta certo per mano il programmatore nuovo all'ambiente Mac: è un testo da usare come riferimento una volta che si siano appresi i primi rudimenti sul Toolbox o quando, già esperti, si ha bisogno di conoscere certi particolari che è impossibile ricordare.

Si sentiva perciò la mancanza di un testo che aiutasse i potenziali programmatori nei primi approcci. Il documento che state leggendo vuole coprire, seppur in via provvisoria, questa lacuna della documentazione.
A questo scopo centreremo il nostro interesse sui diversi passi necessari per sviluppare un programma. Utilizzeremo come esempio una semplice applicazione che svilupperemo in differenti linguaggi. Questa applicazione tocca praticamente tutte le caratteristiche fondamentali di Macintosh e del suo particolare ambiente.

Il seguito di questo documento è organizzato come segue:

2 **Gli ambienti moderni**. Una descrizione discorsiva delle peculiarità dell'ambiente Macintosh e degli stumenti messi a disposizione dal Toolbox.

3 **L'applicazione**. Descrizione, a prescindere dalla sua implementazione su Mac, della struttura e delle caratteristiche della applicazione usata come esempio.

4 **Il Toolbox**. Le singole parti del programma di esempio vengono analizzate per descrivere come utilizzare le varie parti del Toolbox e farle interagire correttamente.

5 **I sistemi di sviluppo**. In questa parte del documento vengono trattati i sistemi di sviluppo utilizzati. In particolare verranno descritte le differenti caratteristiche dei linguaggi e l'ambiente di sviluppo fornito dai vari sistemi (editors, compilatori, linkers, assemblatori e tools di sviluppo).

2 Gli ambienti "moderni"

"Il nostro Eroe si aggirava per la tenebrosa Selva dei Modi, alla ricerca dell'Opzione Perduta. La sua mente vagava tra i ricordi del Grande Manuale alla ricerca di una scorciatoia nell'Albero Gerarchico dei Comandi, tra gli Inserimenti e le Cancellazioni, ma tutto sembrava vano... La Domanda Eterna lo attendeva implacabile, il Cursore occhieggiava dal Video, in attesa... «Copy from what file? »"
(Da "Editors", S/w Production).

Ok, facciamo l'appello: chi non si è mai avventurato per le inestricabili foreste dei Modi*, tra opzioni di Insert, Find, Delete, ecc., talvolta perdendo l'orientamento (e spesso la pazienza...) ?
E' una esperienza quasi quotidiana non solo per chi programma, ma anche per chi vorrebbe scrivere un documento o dei semplici appunti.
Chi in vita sua non ha mai dovuto "digerirsi " almeno un paio di Manualoni, note e richiami a piè di pagina compresi?
Eh, sì! Tutte le volte la stessa storia: si cambia ambiente, e si cambiano anche le regole di comunicazione!

Dal "Dialogo di un Utente con il suo Personal":
"Perchè quando sto usando il tuo Word Processor, il Delete (tasto, n.d.r.) cancella il carattere a sinistra del cursore, mentre quando uso il Foglio Elettronico, cancella il carattere sottostante?" "Perchè chi ha scritto i due programmi non seguiva uno standard preciso, ma solo la sua predisposizione personale; inoltre lo schiacciare Delete non dice a che cosa deve essere applicata la funzione richiamata, ma sottintende solo che si debba fare riferimento al carattere sotto al cursore. Ma quest'ultimo deve sovrascrivere il precedente o viceversa essere sovrascritto?..."

* : "Un modo di un sistema interattivo è uno stato dell'interfaccia utente che dura per un certo periodo di tempo, non è associato ad alcun oggetto particolare, e non ha altro ruolo che dare una interpretazione all'input dell'operatore"(Larry Tesler). In pratica i modi più comuni sono quelli di Inserimento o quelli di Ricerca, in cui per lo stesso carattere della tastiera esistono interpretazioni diverse a seconda del modo durante il quale viene inviato.

Le citazioni introducono un breve discorso sugli ambienti di programmazione, come sono tradizionalmente e come invece dovrebbero essere, alla luce degli ultimi studi sulla comunicazione e sulle interfacce uomo-macchina.

Negli ambienti di sviluppo tradizionali, la comunicazione tra uomo e macchina avviene secondo uno stile che potremmo definire *verbo-oggetto*, nel senso che qualsiasi interazione avviene in due momenti successivi:

a) l'utente invia un comando al programma, richiedendo l'esecuzione di una determinata funzione resa disponibile dal programma stesso, e poi

b) indica lo o gli oggetti su cui tale operazione dovrà essere eseguita.

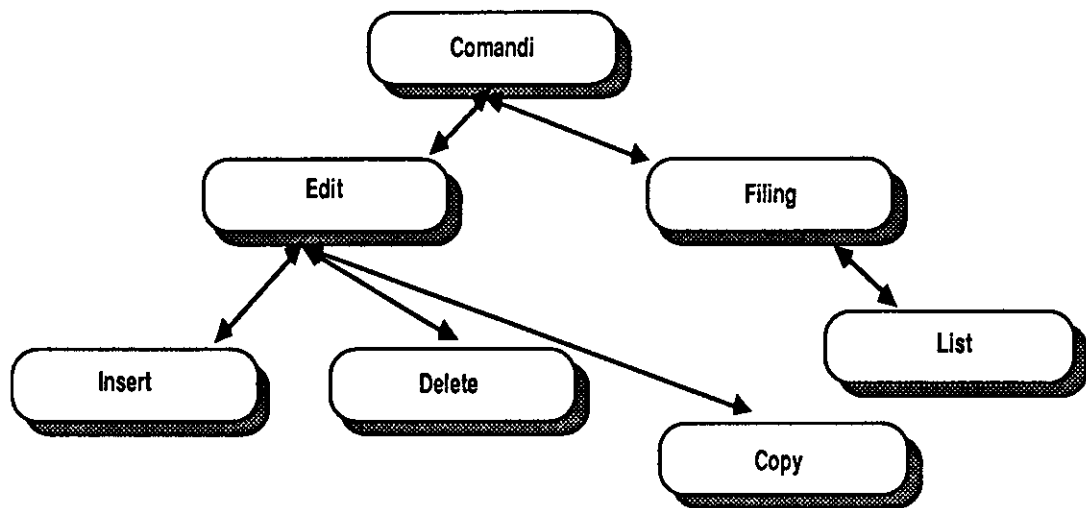
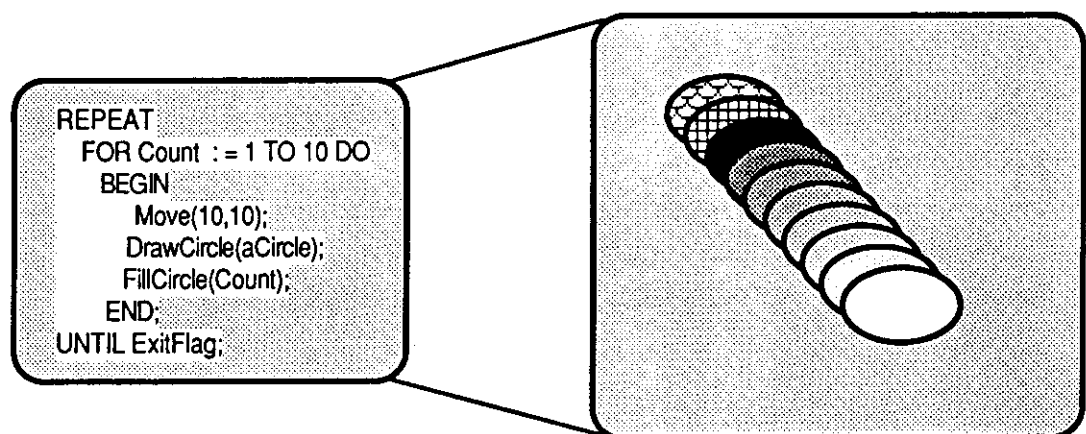
Per esempio (v. il brano precedente) se si vuole inserire, trovandosi in un editor tradizionale , la copia di una parte di un file all'interno del testo che si sta manipolando cosa è necessario fare?
Bisogna richiedere , se disponibile, l'opzione di copia con un comando (in genere C)opy), cui segue la richiesta da parte del programma del nome del file da cui si intende copiare.
A questo punto se si ricorda il nome esatto del file tutto procede altrimenti nel caso sfortunato (ma non infrequente) che ci si dimentichi tale nome, occorre:

- uscire dalla fase di Copy

- uscire dalla fase di Editing

- entrare nella fase di Filing

- richiedere il listato della direttrice corrente

- ... e rifare il percorso inverso come è mostrato nella figura seguente.

Ma oltre a questi problemi di comunicazione, gli ambienti tradizionali creano difficoltà anche durante la fase di testing dei programmi.
Se per esempio vogliamo verificare il funzionamento di un programma di grafica dobbiamo far eseguire il programma che ci mostrerà sullo schermo un risultato.
Se questo non è quello che volevamo ottenere dovremo entrare nelle fase di editing del programma che ha creato l'errore togliendo così dallo schermo l'evidenza dell'errore stesso, mentre invece ciò che vorremmo é la situazione seguente in cui sono presenti sia la rappresentazione grafica sia il codice che la produce.

Ciò accade perchè in genere i sistemi operativi tradizionali dei PC non consentono di operare su più finestre, ma gestiscono il video in modo indivisibile.

Entrambi questi tipi di problemi vengono superati nei cosiddetti *Ambienti Integrati.*
In tali ambienti vale la convenzione che in ogni momento sono disponibili tutte funzioni di sistema, e sono richiamabili su finestre sovrapponibili, attivabili a piacere.
Ciò permette ad esempio di editare un programma e di vedere contemporaneamente i risultati della sua esecuzione su un'altra finestra, permettendo così un più facile debugging.

Ma la convenzione veramente innovativa è il ribaltamento dello schema di comunicazione, da verbo-oggetto a *oggetto-verbo,* allo scopo di avvicinare il rapporto dell'uomo con la macchina al livello delle sue normali interazioni con il mondo reale; in altre parole, in questi ambienti si cerca di simulare la vita reale, piuttosto che imporre diverse convenzioni.
Le operazioni avvengono ancora in due fasi, ma la prima ora è quella di *selezione* degli oggetti su cui fare le manipolazioni, e solo in seguito viene indicata l'*azione* da portare a termine su tali oggetti.
La selezione avviene per mezzo di un nuovo meccanismo hardware, il mouse, che permette all'utente di indicare l'oggetto da selezionare, invece che scriverne le caratteristiche (ad esempio, con il mouse si percorre e si evidenzia la parte di testo da trasferire,

Questo è un testo di prova in cui la la parte scura è quella selezionata

mentre tradizionalmente occorre marcarne l'inizio e la fine con movimenti di cursore e comandi inviati dalla tastiera).
Un indicatore, spesso una freccia come la seguente,

segue ad ogni istante il movimento che l'utente fa compiere al mouse, consentendogli di operare su tutto lo schermo, indipendentemente dalla finestra selezionata. (Ciò consente tra

l'altro di cambiare la finestra su cui si opera correntemente).
Un'ulteriore innovazione è data dall'introduzione della funzione di **Undo**, che annulla l'ultimo comando inviato, allo scopo duplice di proteggere l'utente da sviste e di evitare le richieste di conferma dei comandi (poichè in caso di errore comunque la situazione è recuperabile).
La soluzione rappresentata dagli *Ambienti Integrati* riduce (quasi annulla) la necessità di manuali particolareggiati per le diverse applicazioni: se viene rispettata in ognuna di esse la convenzione sul linguaggio di comunicazione adottata nell'Ambiente Principale (quello cioè in cui le Applicazioni "girano"), all'Utente non serve altro che sapere che cosa sia permesso fare e quali conseguenze abbia, e non più come chiedere al programma di farlo.
La convenzione utilizzata negli *Ambienti Integrati* permette di simulare la vita reale, in modo tale da non richiedere uno sforzo di adattamento ad un ambiente diverso da quello quotidiano.
E, come nella vita reale le nostre azioni si basano sulla regola oggetto-verbo (difficilmente vedrete un fumatore accendere il fiammifero prima di avere estratto la sigaretta dal pacchetto), così negli *Ambienti Integrati* si adotta la convenzione secondo cui prima si seleziona l'oggetto da manipolare, e poi il metodo di manipolazione.
Un esempio dei vantaggi che ne derivano viene dall'utilizzo di un Data Base Manager: tradizionalmente occorrerebbero tre diverse routine di cancellazione:

cut word

cut field

cut record

che in un Data Base *Oggetto-Verbo*, sarebbero tutte assorbite da un'unica routine CUT, che verrebbe applicata all'oggetto correntemente selezionato, indipendentemente dal fatto che esso sia una parola, un campo o un intero record.
Il problema nuovo che sorge dopo la scelta di operare in un *Ambiente Integrato* è la costruzione di un programma che vi si inserisca correttamente.
Perchè rispetti la convenzione deve essere dotato di un'interfaccia utente che abbia un comportamento simile, se non

uguale a quello "globale", un'interfaccia cioè molto sofisticata, con la caratteristica di permettere di richiamare in ogni momento le funzioni disponibili nell'ambiente.
Allo scopo di semplificare la vita al programmatore sono allora nati i cosiddetti Software Frameworks (intelaiature), che sono insiemi di procedure che permettono una facile comunicazione tra un'applicazione e il sottostante sistema operativo.
Si supera in questo modo la grave difficoltà del costo che verrebbero ad avere programmi integrati completamente riscritti. Chi pubblica il software allora è interessato dalla possibilità di mettere sul mercato un prodotto di buona qualità a prezzo accessibile, oltre che altamente competitivo; chi acquista ha il vantaggio di spendere poco (!) e quello di non doversi "digerire" lunghe pagine di istruzioni, perchè le regole di comunicazione sono già sottintese e comuni a tutti i programmi che "girano" sulla macchina in suo possesso.

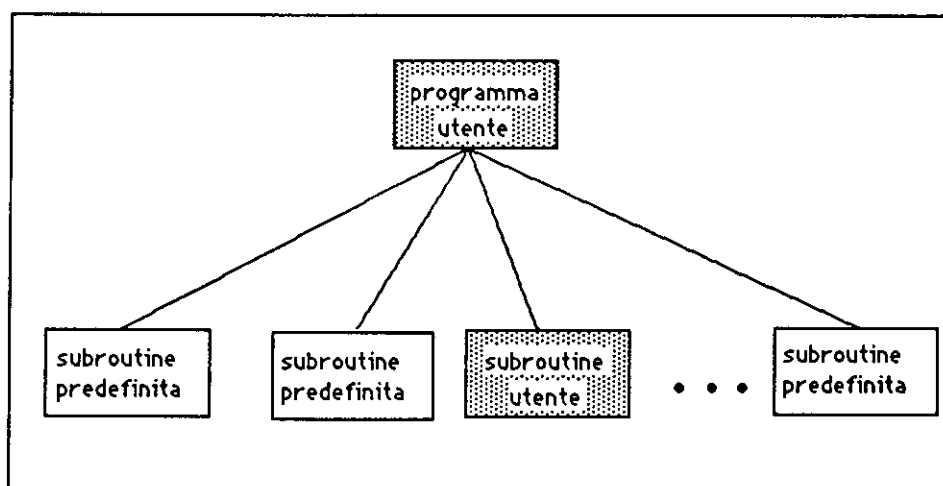



Figura a). Creazione di un programma applicativo utilizzando una libreria di subroutines.

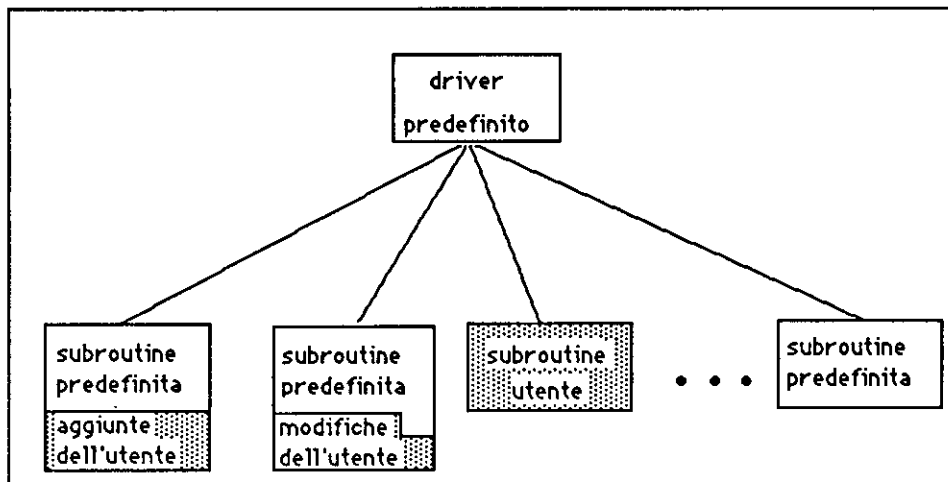



Figura b). Creazione di un programma usando una intelaiatura software.

L'ideale ambiente di programmazione dovrebbe allora essere quello di fig. b), in contrapposizione a quello in fig. a), che rispecchia ciò che finora è in commercio.
La differenza sta nel fatto che il driver dell'applicazione non è più a carico del programmatore, ma viene fornito in partenza dal sistema. In tal modo il programmatore non deve più occuparsi di come "aggregare" le varie parti del sistema, ma solo di specificare quali debbano essere le risposte del programma alle diverse richieste dell'utente.
Ciò si ottiene o modificando delle subroutine predefinite o aggiungendo piccole quantità di codice al sistema.

"La notte era serena, ma senza luna. In lontananza si intuivano le forme della Grande Montagna di Carta, che attendeva di essere scalata. L'Eroe, era alla ricerca della Parola Magica, che gli permettese di mandare il suo File Messaggio sulla Montagna, e di ottenerne l'agognata stampa...
All'improvviso, ecco il Comando!

nroff -me -Tr32 Messaggio | lpr

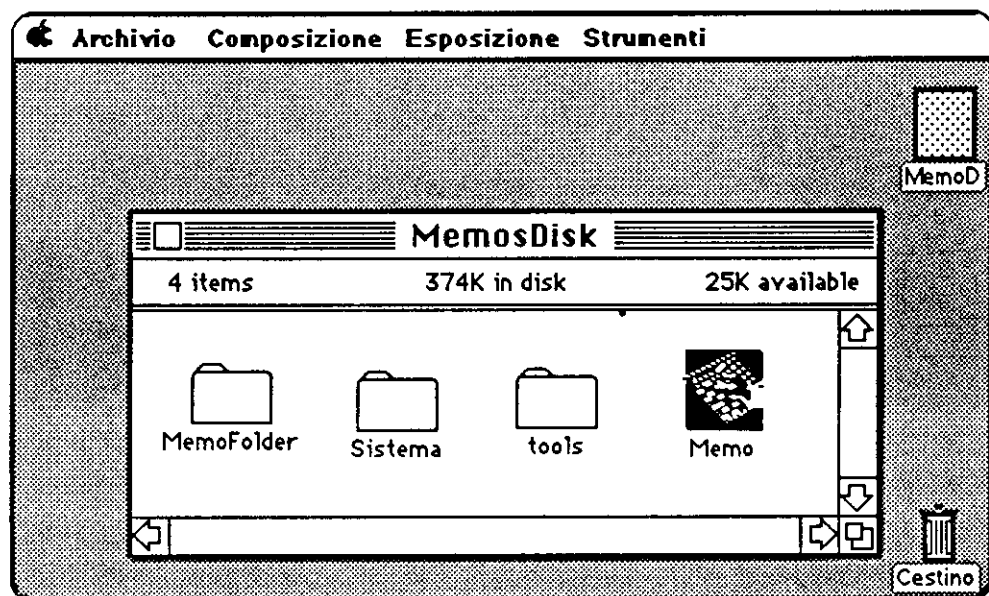
Egli lo inviò con grande velocità, ma appena premuto <Return>, cadde nella più profonda tristezza: si era dimenticato il Segno del Tempo Perduto, quel "&" che gli avrebbe permesso di fuggire, mentre il Messaggio arrivava alla Montagna..."

Già, brutta faccenda, vero? Ci si dimentica un "&" e subito si è inchiodati lì finchè tutto non sia finito. Ammesso che vi siate ricordati tutto il comando in modo corretto...
Quella vista nel brano precedente era la versione mitologica dell'interfaccia a comandi tipica del S.O. UNIX™.
L'utente di UNIX, avendo a disposizione un numero molto vasto di comandi, trova difficoltà nel ricordarli tutti, e tutti con la loro particolare sintassi delle opzioni. Inoltre un singolo carattere può assumere un ruolo decisivo, come abbiamo visto, e una svista su un solo carattere può capitare a tutti, no?...

Altri due tipi di interfacce sono quelle a menù e quelle a Icone. Di queste, il primo tipo rappresenta certamente un miglioramento rispetto al tipo a comandi, in quanto permette di scegliere tra più operazioni all'interno di un gruppo che viene presentato sullo schermo,

1 ⇔ Mostra gli accessori
2 ⇔ Apri l'archivio
3 ⇔ Seleziona i Menù
4 ⇔ Usa la composizione
5 ⇔ Scegli gli strumenti

Seleziona una delle opzioni con il numero corrispondente.

ma impone di conoscere il significato delle singole opzioni (che in generale sarà diverso da applicazione ad applicazione); inoltre, spostarsi da un menù all'altro non è certamente un passatempo dei più graditi. Un esempio di questo tipo si può trovare nei comuni fogli elettronici, in cui il menù "/" funge da radice dell'albero dei comandi.
Il terzo tipo è sicuramente il più avanzato, quello con cui è più semplice colloquiare, perchè utilizza la rappresentazione grafica dell'oggetto o dell'azione,

e non il nome (simbolico o no); perciò ha il vantaggio di non richiedere un lungo periodo di apprendimento per acquisirne appieno il significato.
Sembra abbastanza ovvio che per ordinare al sistema operativo la cancellazione di un File da disco, è più semplice selezionare la rappresentazione grafica del file e indicare al S.O. di infilarlo nel cestino sotto la scrivania,

piuttosto che inviare il comando

rm <File>

(interfaccia a comandi) oppure selezionare il menù File, e scegliere tra le diverse opzioni (Retrieve, Save, List, Erase, ecc.) che vengono presentate.
Il personal computer acquista una nuova efficacia con l'introduzione del linguaggio iconico, con cui cioè la trasmissione

di informazioni tra uomo e macchina si svolge per immagini, e in cui l'uomo, per mezzo del mouse, è in grado di mostrare (in parte simulandolo) ciò che vuol fare, piuttosto che essere costretto a "trovare le parole per dirlo".
In conclusione ciò che gli *ambienti integrati* vogliono essere è una ***metafora della vita reale*** in cui gli oggetti vengono prima selezionati, poi manipolati per mezzo di procedure sempre comunque disponibili in ogni momento.

Il nostro strumento

Un'ottima approssimazione di tutto quanto visto in precedenza è data dall'ultimo nato in casa Apple: MacIntosh.

MacIntosh (Mac per gli amici) è un potente personal computer, dotato di:

⇒ Microprocessore MC 68000, architettura a 32 bit e clock a 8MHz.

⇒ Memoria di 512 K RAM (espandibile fino a 1 M) e 64K (128 K) ROM (vedremo poi quanto importanti!)

⇒ Dischi da 3.5 pollici Sony da 400K (800K)formattati

⇒ Video 9 pollici Bianco/Nero di 512x342 punti bit mapped

⇒ Interfacce: 2 porte seriali RS 422/423
(porta SCSI)
drive aggiuntivo
mouse ad un bottone

⇒ Output audio con generatore a 4 voci

⇒ Tastiera di 58 caratteri, mappati via software

⇒ Orologio (h,m,s,mm,gg,aa) a batteria

MacIntosh è stato creato seguendo proprio le indicazioni che

abbiamo prima esaminato, tenendo conto che deve rivolgersi anche a chi non sa (e non vuole) programmare, ma vuole avere un approccio il più amichevole possibile con la macchina. A tale scopo viene richiesta ai programmatori di nuove applicazioni su Mac la coerenza con il sistema globale anche all'interno dei loro programmi. Perciò Mac fornisce al programmatore un insieme di supporti software che gli permettano di seguire le linee tracciate, senza però dover "reinventare" tutto da capo.

Il Software

Numerosi sono gli strumenti per lo sviluppo su Mac a disposizione:

⇒ per i programmatori assembler vi sono "**68000 Development System - Apple**" e "**MacASM - MainStay**"

⇒ chi vuol continuare ad utilizzare il BASIC, può utilizzare il "**BASIC 2.0 - MicroSoft**", oppure trasportare programmi già sviluppati sotto MSDOS per mezzo del "**PC Basic Compiler**" della Pterodactyl Software ;

⇒ gli appassionati al C hanno a disposizione "**Aztec C68K-c**" della Manx Software Systems, oppure "**Softworks C**" della Softworks Ltd., "**Hippo-C**" della Hippopotamus, il "**Megamax C**" o infine "**Mac C**" della Consulair;

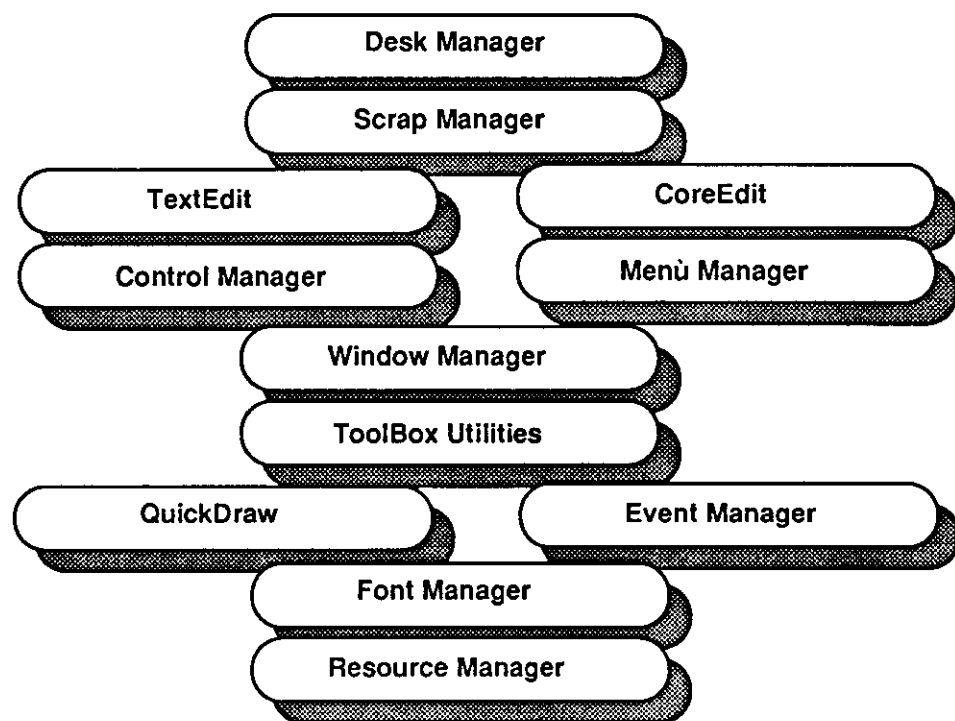
⇒ per ciò che riguarda il Pascal sono disponibili: "**Workshop Pascal**" che gira su Lisa,"**MacAdvantage: UCSD Pascal**" della Softech Microsystems, e l'interprete "**Macintosh Pascal**" della Apple; inoltre saranno presto disponibili i compilatori pascal della **TML** e della **OnStage**.

⇒ inoltre sono a disposizione i linguaggi tradizionali, come il "**MacCOBOL**" della Micro Focus, il "**MacFortran**" della Absoft e il "**Fortran77**" della Softech;

⇒ o linguaggi di recente sviluppo, come "**MacForth**" della Creative Solutions, "**Modula-2**" della Volition Systems, "**MacModula-2**" della Modula Corporation, "**Neon**" della Kriya Systems;

⇒ un posto a sè meritano gli ambienti "**ExperLisp**" e "**ExperLogo**" della Expertelligence.

L'ambiente Macintosh

L'ambiente in cui si muove un programmatore di Mac è costituito da una memoria RAM che può raggiungere i 512K bytes e da 64K bytes di codice in ROM così strutturato:

Questo codice implementa già tutte le procedure per la gestione degli oggetti utilizzabili in ambiente Mac. Vale a dire:

grafica	Quickdraw
menù	Menu manager
finestre	Window manager
controlli	Control manager
eventi	Event manager

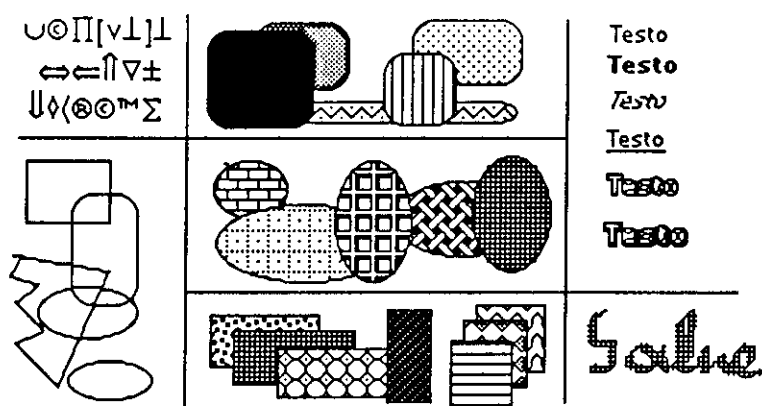
risorse	Resource	manager
finestre di dialogo e avvertimento	Dialog	manager
fonti	Font	manager

inoltre contiene le routine di gestione della memoria e i driver di interrupt (mouse, tastiera, dischi, porte seriali).

QuickDraw: è l'insieme delle procedure (e funzioni) implementate in ROM che permettono la gestione dello schermo grafico di Mac. QD permette di dividerlo in più aree indipendenti, in cui si possono disegnare:

⇒ caratteri di testo, in più fonti, proporzionali, con stili diversi, grassetto, corsivo, sottolineato, ecc.

⇒ linee, rettangoli, ovali, ecc.

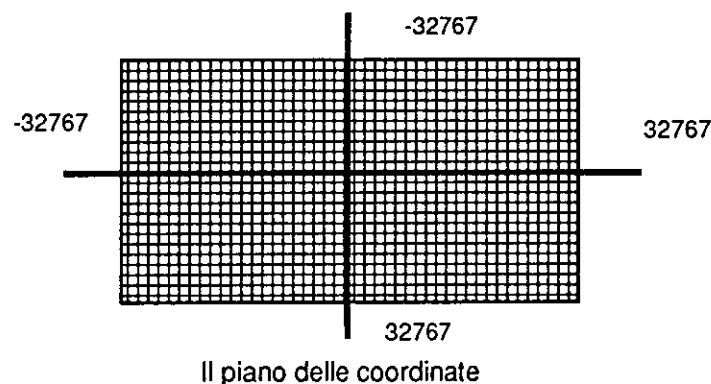
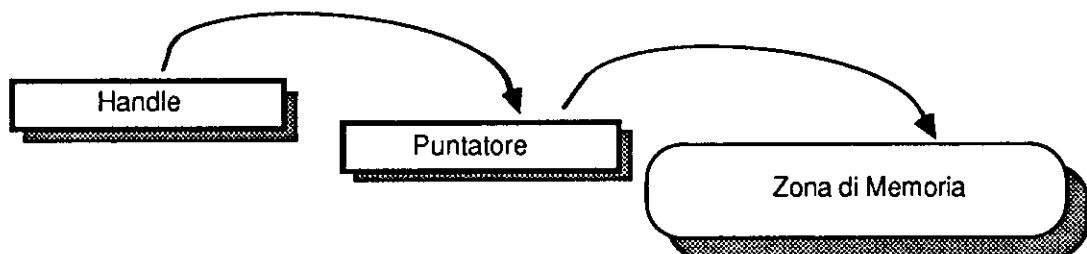
⇒ regioni di forma arbitraria, poligoni, ecc.

Inoltre:

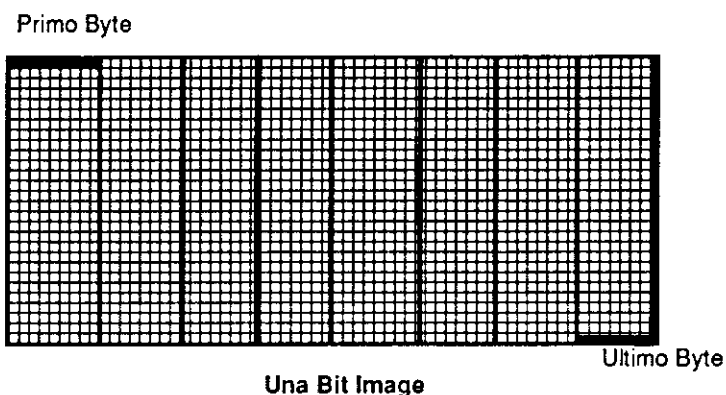
⇒ permette di definire più "porte" sullo schermo, ognuna indipendente e dotata di proprie coordinate, penna, ecc.

⇒ esegue il clipping automatico nelle regioni definite.

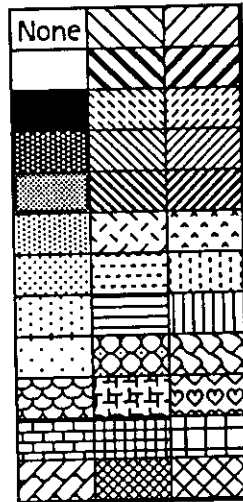
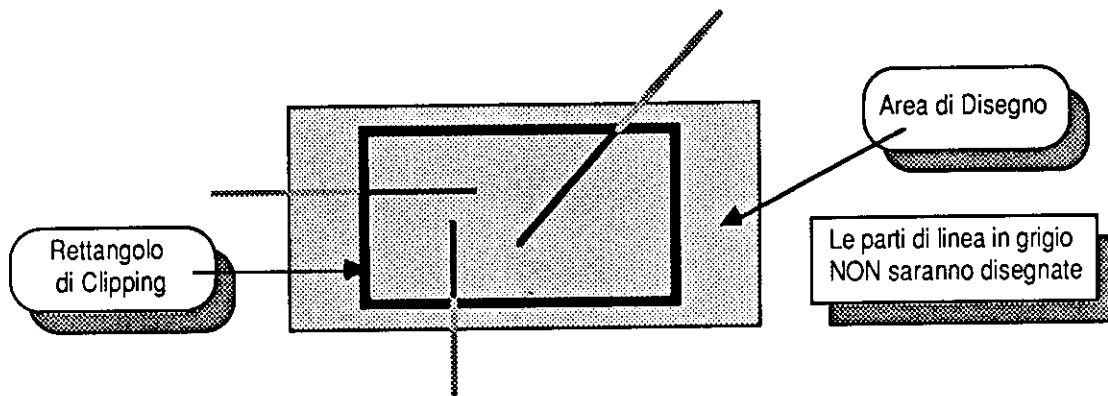
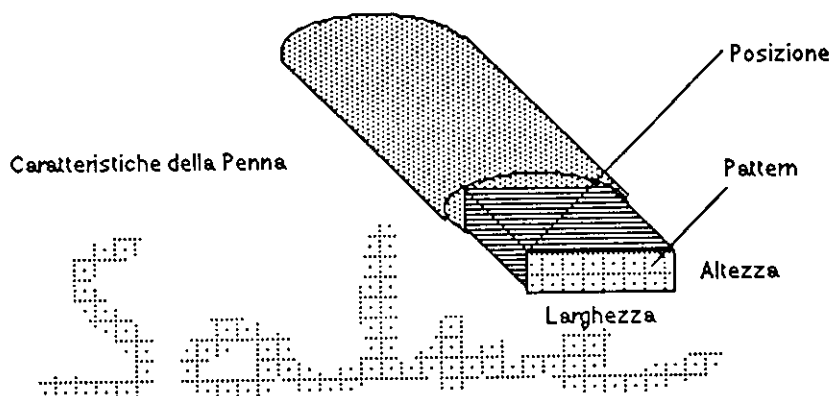
⇒ consente di disegnare al di fuori dello schermo, di preparare cioè delle figure che verranno mostrate solo in seguito.

Le routines di QD (come di tutti gli altri manager che vedremo in seguito) possono essere richiamate direttamente all'interno di programmi Assembler o Pascal; sarà poi compito del compilatore (assemblatore) di collegare alle routine in ROM queste chiamate . Tutte le routine di QD fanno riferimento a un piano cartesiano, in cui ogni quadrante ha dimensioni 32768 x 32768.

Il piano delle coordinate

Le entità geometriche usate sono quella di punto, definito come una coppia di interi (o come LongInt), di rettangolo, definito per mezzo della coppia di vertici (punti) superiore sinistro e inferiore destro, e di regione, definita per mezzo delle routine di QD necessarie a tracciarne i bordi.
Tutti gli oggetti di QD (e degli altri manager) sono accessibili attraverso Handles, cioè puntatori a un puntatore principale, che a sua volta punta alla zona dati relativa.

Ciò accade perchè Mac gestisce autonomamente la memoria, collettando le zone non più utilizzate.
Nel caso in cui, per esempio, si modifichi una regione, questa viene completamente riscritta, per cui il puntatore principale viene ad essere modificato e spostato in memoria.
Se ne esistessero più copie, ovviamente queste ultime non punterebbero più ai dati corretti.
In parallelo con le entità geometriche, in QD sono definite delle entità grafiche, che hanno relazione stretta con le precedenti, ma sono fisicamente rappresentabili.
Esse sono la Bit Image, la BitMap, il pattern e il cursore.
La Bit Image è un vettore di bit, che ha in memoria rappresentazione lineare.
Tale vettore può essere pensato anche come una matrice, righe per colonne, in cui il numero di byte per ogni riga è detto larghezza di riga.

Una Bit Image

Quando si combinano i due aspetti di un'immagine e si considerano sia l'entità fisica di una immagine, sia quella logica di piano cartesiano e di rettangolo, si ottiene una BitMap. Una BitMap è definita come record di tre informazioni: un puntatore ad una Bit Image, una larghezza di riga e un rettangolo su cui l'immagine giace e da cui la BitMap prende dimensioni e coordinate.

Un cursore è un'immagine di 16x16 bits, che rappresenta la posizione del mouse sullo schermo, di cui seguono alcuni esempi.

mentre un pattern è un'immagine di 8x8 bits, che viene usata come "colore" nei disegni, negli sfondi, ecc.

L'ambiente in cui vengono effettuate tutte le operazioni grafiche è detto Grafport o porta grafica.
Le porte grafiche sono le strutture sulle quali i programmi costruiscono le finestre, fondamentali per l'interfaccia utente Mac.
Le porte grafiche sono anch'esse definite in Pascal come record, e contengono le informazioni sul dispositivo di output cui la porta è associata, la BitMap su cui operare ed il rettangolo, sottinsieme della BitMap, usato dalla porta.
Altre due informazioni riguardano la regione visibile e la regione di clipping, cioè quella all'esterno della quale il disegno non viene effettuato.

Inoltre vi sono le informazioni sui pattern usati (background e foreground), su tipo, posizione, ecc. della penna con cui disegnare, fonte, stile, dimensioni, ecc. dei caratteri di testo.

Ogni porta grafica ha un proprio sistema di coordinate locali, definito attraverso il rettangolo della BitMap associata.
Tutte le informazioni all'interno di una PG sono relative a questo sistema.
Quando viene creata una nuova PG, la sua BitMap viene fatta puntare all'intero schermo, e sia il rettangolo della BitMap, sia quello della porta vengono definiti di 512 x342 bit, con il punto (0,0) coincidente all'angolo superiore sinistro dello schermo.
L'origine degli assi coordinati può essere modificata per mezzo della procedura SetOrigin(x,y), dove (x,y) è il nuovo punto (0,0) espresso nelle vecchie coordinate.

Ogni operazione grafica avviene:

⇒ sempre all'interno di una PG, sulla Bit Image e con il sistema di coordinate definito dalla BitMap della porta.

⇒ sempre nell'intersezione del rettangolo della BitMap, con il rettangolo di definizione e clippata alla regione visibile e a quella di clipping.

⇒ sempre nel punto in cui è posta la penna.

⇒ normalmente con le caratteristiche della penna della PG.

Con le routines di QD si possono disegnare linee, figure (rettangoli, con o senza angoli arrotondati, ovali, poligoni, regioni) e testi di ogni tipo con diverse fonti e diversi stili come si può vedere di seguito

Caratteri Helvetica Normali
Caratteri Helvetica Bold
Caratteri Helvetica Italici
Caratteri Helvetica Sottolineati
Caratteri Helvetica Contornati
Caratteri Helvetica Ombreggiati
Caratteri Helvetica Italici + Bold + Sottolineati
Caratteri Helvetica Ombreggiati + Sottolineati
Caratteri Times Normali
Caratteri Times Bold
Caratteri Times Italici
Caratteri Times Sottolineati
Caratteri Times Contornati
Caratteri Times Ombreggiati
Caratteri Times Italici + Bold + Sottolineati
Caratteri Times Ombreggiati + Sottolineati

.........e in molte altre fonti

L'**Event Manager** di Mac è il legame tra i programmi e l'utente. Infatti il sistema operativo di Mac associa ad ogni azione esterna un Evento, che viene comunicato al programma per i suoi usi interni. Una applicazione tipica di Mac (e lo vedremo in seguito) è guidata dagli eventi, cioè il programma rimane in attesa che un Evento gli venga segnalato e risponde in maniera appropriata.
Gli eventi sono di vario genere, a seconda della loro origine e del loro significato. I più importanti sono quelli che registrano le azioni dell'utente:

⇒ gli Eventi mouse down e mouse up vengono segnalati quando l'utente schiaccia o rilascia il bottone del mouse.

⇒ gli Eventi key down e key up segnalano la pressione o il rilascio di un tasto della tastiera o del tastierino numerico. Inoltre viene automaticamente segnalato l'evento auto key, quando l'utente tiene un tasto premuto.

⇒ l'evento disk inserted, che viene segnalato quando un disco viene inserito in uno dei due drive.

⇒ un abort event viene quando viene premuta una combinazione speciale di tasti (generalmente Command-punto).

Altri eventi sono generati dal Window manager, per gestire le finestre sullo schermo:

⇒ un activate event segnala quando una finestra attiva diventa inattiva o viceversa. Normalmente vengono segnalati in coppie (una finestra viene disattivata e un'altra si attiva).

⇒ un update event segnala che il contenuto di una finestra deve essere ridisegnato, di solito in seguito ad aperture, chiusure o attivazioni o movimenti di finestre sullo schermo da parte dell'utente.

Altri due eventi, I/O driver e network, sono utilizzati dal sistema di I/O di basso livello.

In ogni applicazione sono poi definibili fino a quattro tipi di eventi, utilizzabili per qualunque scopo all'interno del programma.
Infine è previsto l'Evento null, che indica come non vi siano per il momento eventi da segnalare.
Tutti gli eventi, una volta generati, vengono messi in coda, in attesa che il programma se ne occupi. Il programma a sua volta può scegliere di rispondere solo ad uno o più tipi di eventi, andandoli a ricercare sulla coda, e alterandone l'ordine. in ogni caso, gli eventi vengono segnalati secondo la seguente priorità:

⇒ activate (prima le finestre che si disattivano, poi quelle attivate)

⇒ mouse down e up, key down e up, disk inserted, abort, network, I/O driver, e eventi dell'applicazione

⇒ auto-key

⇒ update

⇒ null

Gli eventi di attivazione delle finestre, hanno priorità superiore agli altri e vengono conservati in una coda diversa.
L'Event Manager li ricerca in questa coda prima di ogni altro tipo di evento. In questo modo, non ce ne possono essere mai più di due contemporaneamente in coda, uno per la finestra che viene disattivata, e uno per quella che diventa attiva.
Gli eventi vengono segnalati al programma per mezzo di un record, che contiene le seguenti informazioni:

⇒ il tipo di evento;

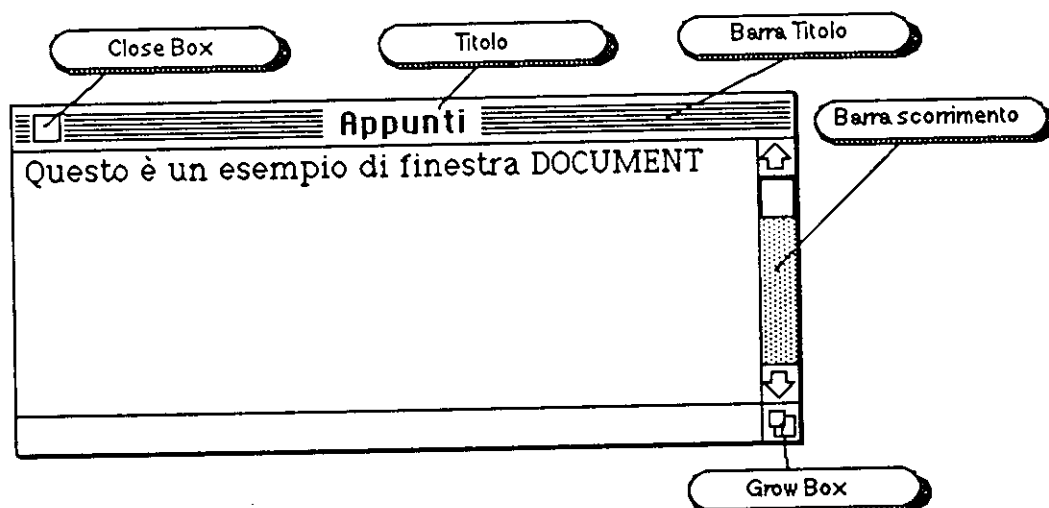
⇒ il momento in cui l'evento è stato segnalato;

⇒ il punto in cui si trovava il mouse;

⇒ lo stato del bottone del mouse e dei tasti modificatori (Shift,

Caps Lock, Option, Command);

⇒ ogni altra informazione necessaria per il tipo particolare di evento.

Le routines per l'utilizzo delle finestre, cioè creazione, modifica, aggiornamento, chiusura, cambio di dimensioni, ecc. sono contenute nel **Window Manager**.
Lo schermo di Mac rappresenta una scrivania, su cui vengono eseguite tutte le operazioni. Esse si svolgono in una o più finestre, di varie dimensioni e formato, e con vari utilizzi.
Tra i tipi di finestre predefiniti, il più comune è quello costituito dalle finestre di documenti.
Esse sono dotate di una barra del titolo, in cui compare centrato il nome del documento in via di preparazione, possono avere una close box (per chiudere la finestra) e una Grow Box, che permette di variarne le dimensioni.

Inoltre possono avere delle barre di controllo, o scorrimento, sia verticali che orizzontali che permettono di controllare lo scrolling del testo contenuto nella finestra.
Altri tipi di finestre verranno visti in seguito e permettono di utilizzare i desk accessories e di comunicare con l'utente.
Per ciò che riguarda il controllo delle finestre sovrapposte,

l'interfaccia utente standard di Mac suggerisce:

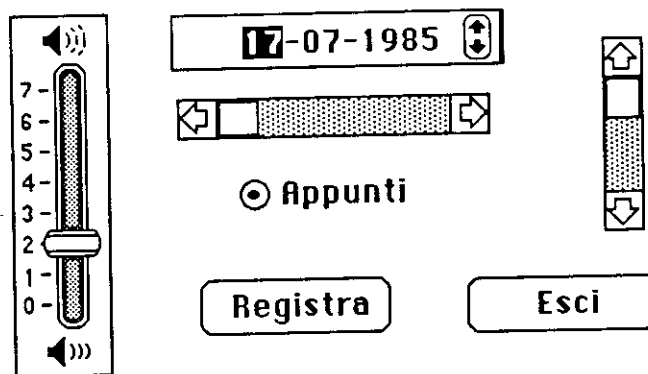
⇒ un click in una finestra inattiva, la rende attiva, portandola davanti alle altre;

⇒ un click nella close box della finestra attiva la chiude o la cancella.

⇒ il dragging (muovere il mouse tenendo premuto il pulsante) nella barra del titolo sposta un profilo della finestra seguendo il cursore, fino a quando il pulsante non viene rilasciato;

⇒ il dragging nella grow box altera le dimensioni della finestra.

Ogni finestra si divide in due regioni: la regione del contenuto, in cui l'applicazione disegna, e l'area della struttura, cioè l'intera finestra.

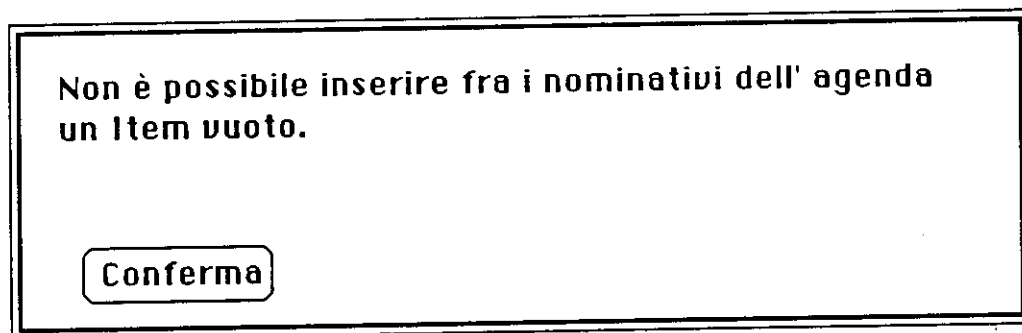
Le routines contenute nel **Control Manager** permettono la gestione delle interazioni con i vari controlli disponibili, tra cui le già viste barre di controllo delle finestre.

Su ogni finestra l'editing dei testi viene fatto per mezzo di **Text Edit**, un gruppo di routine compatte ed efficienti che forniscono le funzioni basilari di text editing e formattazione necessarie in una applicazione. Queste routine svolgono le seguenti operazioni:

- ⇒ inserzione di nuovo testo
- ⇒ cancellazione di caratteri con il tasto di backspace
- ⇒ traduzione delle azioni del mouse in selezione di testo e posizionamento del cursore
- ⇒ movimento del testo all'interno di una finestra
- ⇒ cancellazione del testo selezionato e possibilità di reinserirlo altrove oppure copia del testo senza cancellazione

TextEdit ridisegna automaticamente la parte di testo modificata all'interno della finestra dopo ognuna delle operazioni indicate.

Finestre particolari sono poi le finestre di **dialogo** e quelle di **allarme**: esse permettono di richiedere input particolari all'utente (per esempio il nome di un file da aprire) oppure lo avvertono di situazioni particolari (esaurimento della memoria, impossibilità di effettuare qualche operazione richiesta, ecc.

Non è possibile inserire fra i nominativi dell' agenda un Item vuoto.

Conferma

Il **Dialog Manager** permette di definire con relativa facilità queste finestre particolari e gestisce l'interfaccia con le azioni dell'utente. Un dialog può contenere :

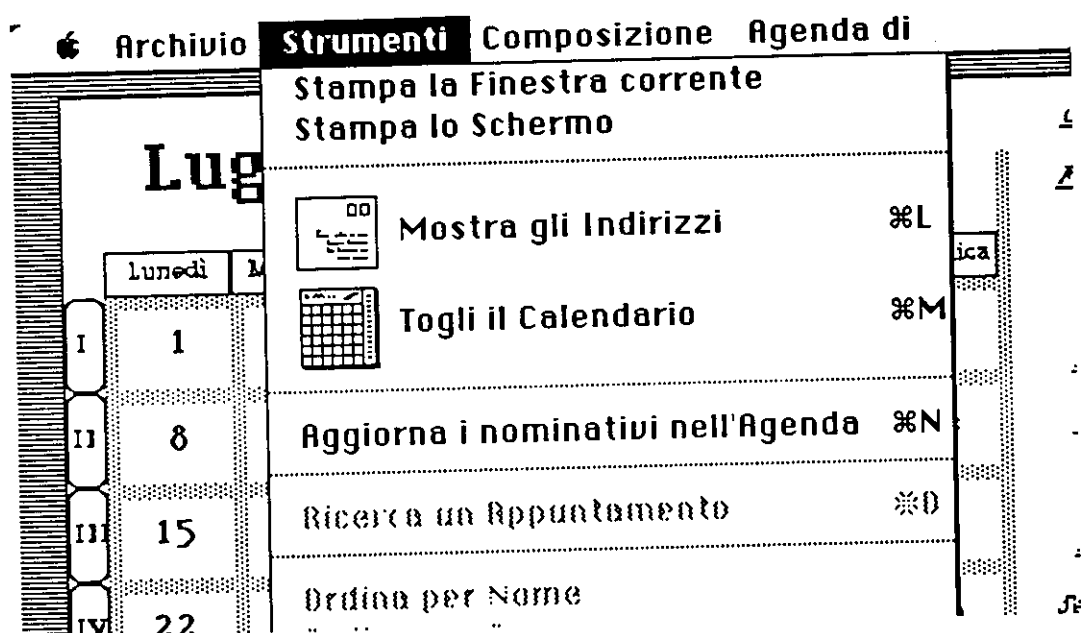
- ⇒ testo informativo
- ⇒ rettangoli in cui si può inserire e modificare del testo (finestre

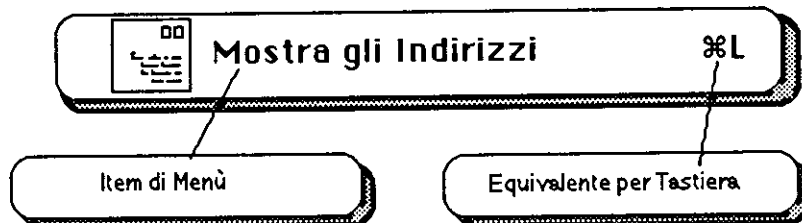
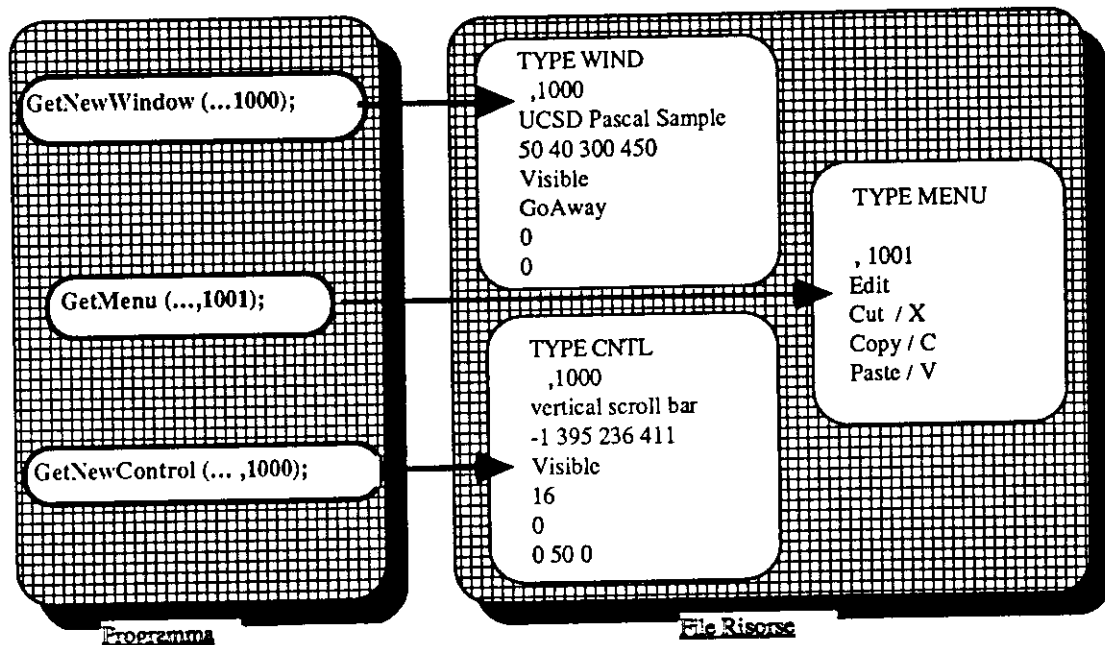
di text edit)

⇒ controlli di ogni tipo

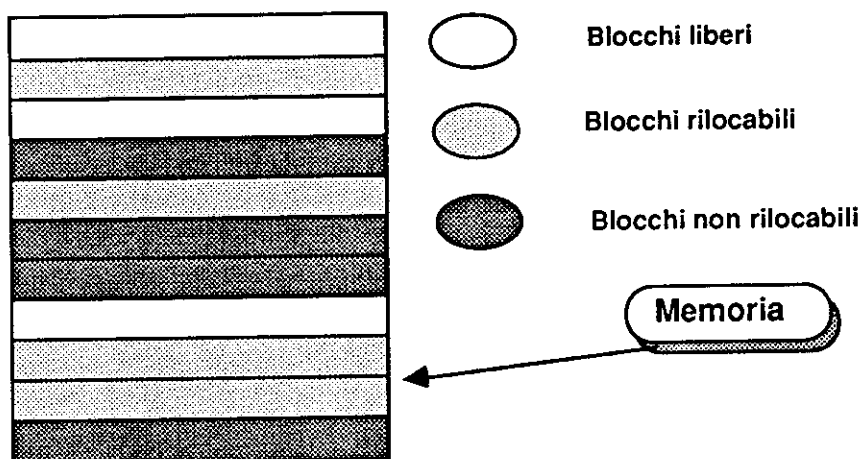
⇒ parti grafiche (icone o "picture" QuickDraw)

⇒ qualsiasi altra cosa definita dall'applicazione

L'uso dei menù, che costituiscono l'altra parte fondamentale dell'interfaccia Mac, è pemesso dalle routines contenute nel **Menu Manager**.
Esse permettono di listare sullo schermo i menù disponibili,

di selezionarli e di far scorrere gli elementi di ognuno di essi, consentendo quindi all'utente di scegliere l'operazione.
Ad ogni elemento di un menù può essere inoltre fornita un "keyboard equivalent", cioè un carattere, che permette di richiamare l'elemento senza andarlo a ricercare ogni volta nel menù, ma solo premendo il tasto Command seguito da tale carattere.
Ogni elemento e ogni menù sono poi disattivabili, in modo da non

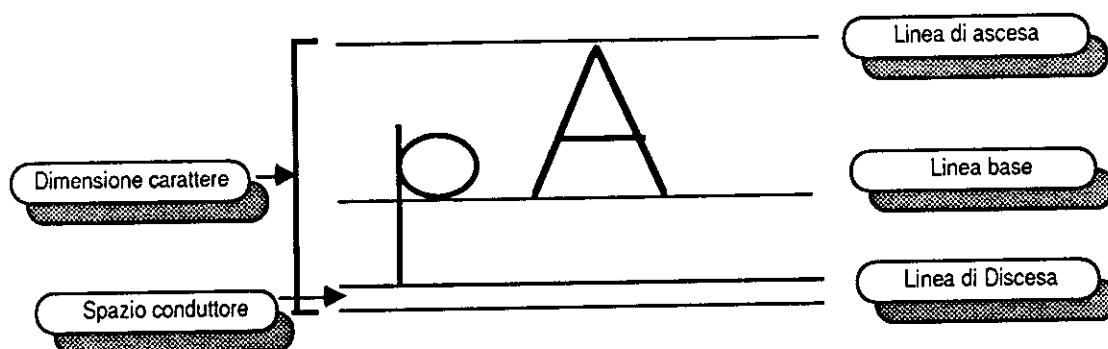
permettere il richiamo di particolari funzioni in momenti in cui non sarebbero corrette.

Infine per il programmatore la grande novità di Mac: l'uso dei **file di risorse** a completamento dei programmi.
Infatti, uno degli aspetti che più rendono difficile la modifica di un programma, e quindi anche il suo adattamento a condizioni diverse da quelle in cui è stato sviluppato (tipicamente la sua "traduzione" in un'altra lingua) è il fatto che i dati sono contenuti, se non tutti almeno in parte rilevante, all'interno del programma stesso.
Ciò comporta un non sempre facile accesso al sorgente del programma, con i rischi che la modifica dei sorgenti comporta.
I programmi per Mac invece fanno sempre riferimento per i loro dati a file esterni, detti Resource File, che vengono creati e modificati in maniera totalmente indipendente.

I dati vengono organizzati per tipi e ad ogni elemento di un particolare tipo viene assegnato un identificatore cui viene fatto riferimento dall'interno del programma.
Il **Resource Manager** inoltre permette anche una gestione efficiente della memoria, in quanto carica in memoria le risorse solo quando queste sono richieste, mentre le salva su disco quando é necessario spazio in memoria. In questo modo viene realizzato, anche se in maniera abbastanza semplice, un meccanismo di memoria virtuale.

Per controllare l'allocazione dinamica dello spazio di memoria è disponibile il **Memory Manager.**
Usando il Memory Manager si possono mantenere aree indipendenti di memoria ed allocare in esse blocchi della dimensione desiderata.
I blocchi di memoria possono essere allocati e rilasciati in qualsiasi ordine, così anzichè variare in modo ordinato lo spazio di memoria tende alla fragmentazione.

I blocchi possono essere rilocabili e non , e nel caso lo siano possono essere bloccati.

Il Memory Manager si occupa di tenere traccia e modificare la situazione della memoria compattandola quando necessario.

Il **Font Manager** è quella parte dell'interfaccia Mac che si occupa di gestire l'uso delle varie fonti, in tutte le dimensioni e tipi, che vengono poi utilizzati da Quickdraw.
Una fonte è un gruppo completo di caratteri con uno stile uniforme come

ABCDEFG...Zabcde...z,;:?./%*+_§&'çè...

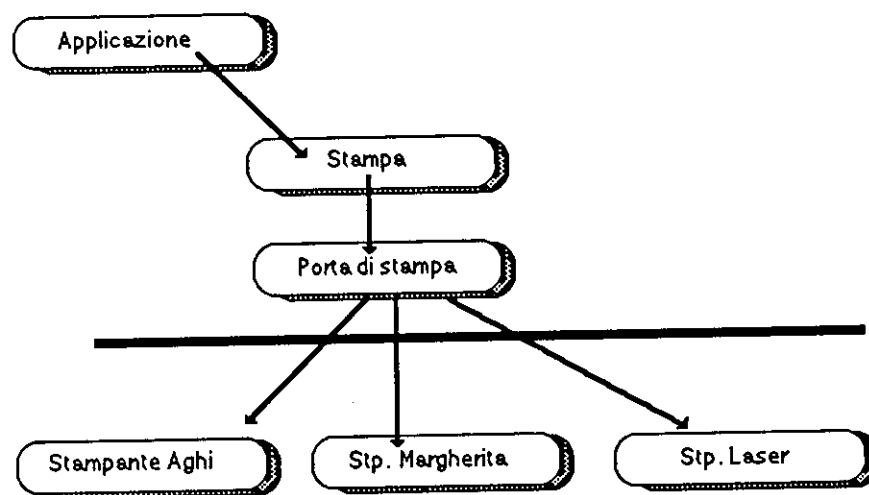
dove ogni carattere è individuato dalle seguenti caratteristiche:

Il **Printing Manager** si occupa di stampare testo o grafica su una stampante.
Esso fornisce:

⇒ due metodi standard di stampa più la possibilità di definirne altri due.

⇒ due dialog per definire il tipo e la qualità della stampa.

⇒ la possibilità di stampare mentre il programma continua a funzionare.

⇒ dei metodi per interrompere la stampa.

Il Printing Manager è disegnato in modo che l'applicazione non si debba occupare del tipo di stampante che é in uso poichè la conversione delle istruzioni di basso livello per comandare i vari tipi di stampante sono fatti in modo automatico.

3 L'applicazione

In questa parte vedremo gli aspetti generali dalla applicazione che utilizzeremo come esempio nel resto del documento. La descrizione riguarda le funzionalità che si vogliono ottenere e l'organizzazione logica del programma, quindi non si parlerà dei dettagli implementativi caratteristici di un determinato linguaggio e neppure dell'uso del Toolbox, che verranno trattati nel seguito.

L' applicazione che abbiamo sviluppato è un semplice indirizzario, un data base a campi fissati con possibilità di ordinamento e ricerca su più campi.

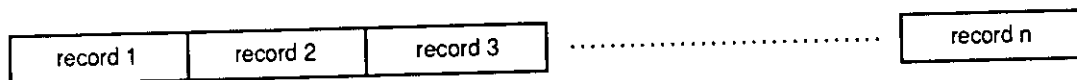
Come prima cosa vediamo quali strutture e tipi di dati verranno utilizzati.

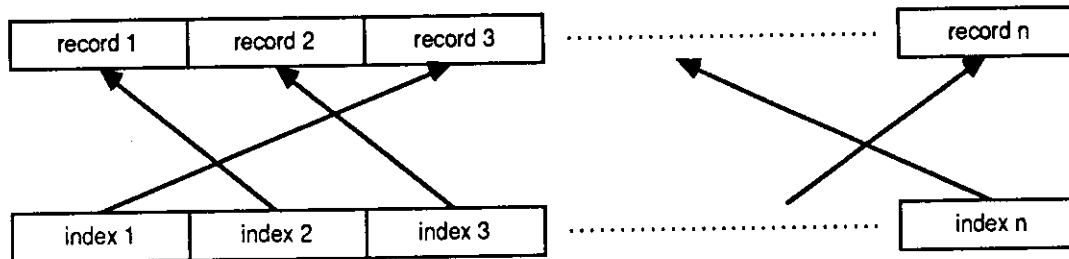
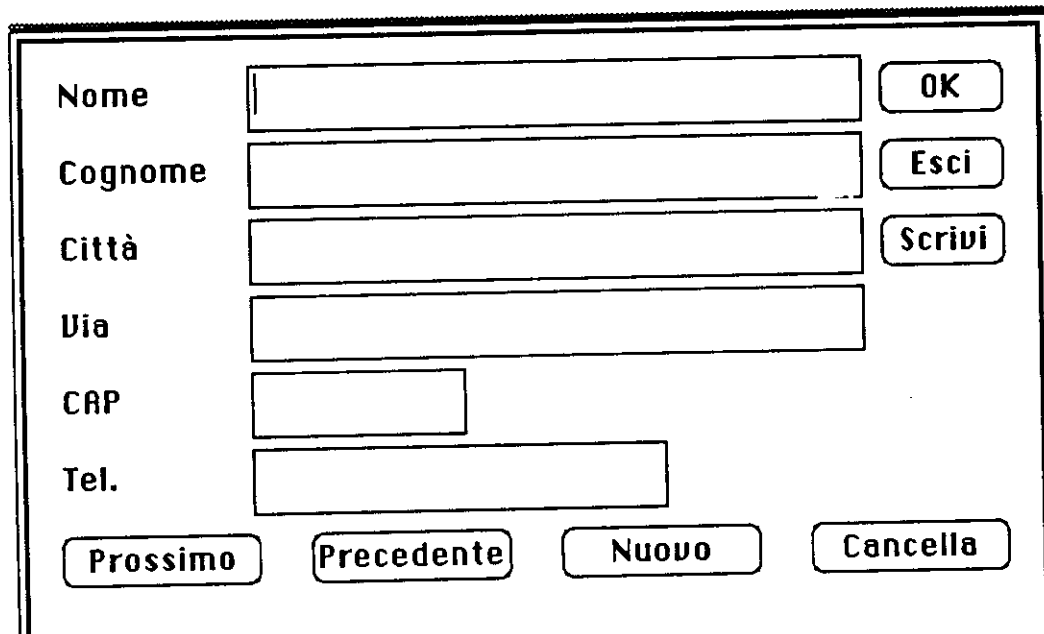
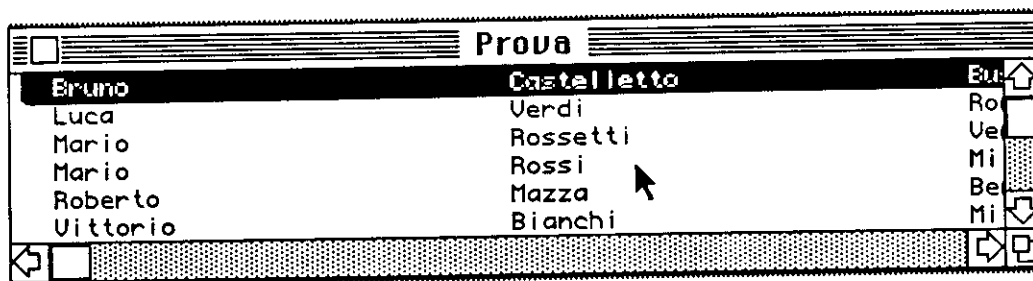
Il tipo di dato strutturato più importante della nostra applicazione è il record che conterrà i dati di un nominativo. I campi di questo record sono 8: nome, cognome, città e via (tutti di 30 caratteri), cap (10 caratteri), telefono (20 caratteri), data, che conterrà l'istante di ultima modifica secondo l'orologio di Mac, e infine un flag che indica se il record è cancellato o meno.

nome	cognome	città	via	cap	telefono	data	flag
30	30	30	30	10	20	4	

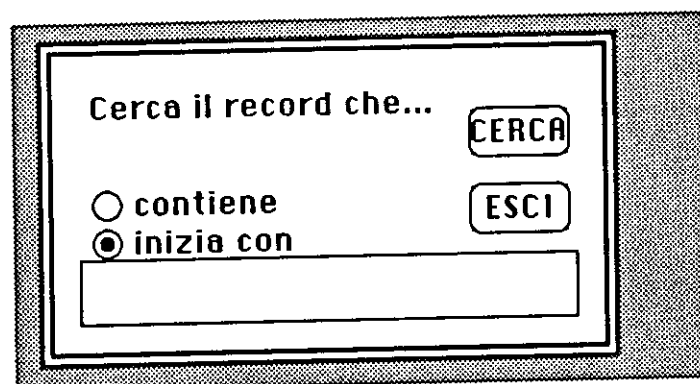
La struttura del record

Tutti i record presenti nel data base sono raggruppati in un array di dimensioni variabili.

record 1	record 2	record 3	....	record n

L' array di record

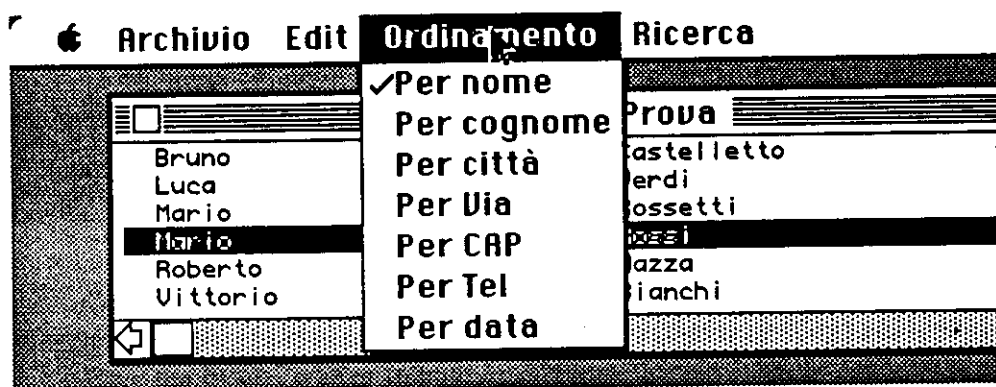
Per gli ordinamenti, invece di ordinare direttamente i record, si utilizzano degli indici. Questi saranno contenuti in un array di dimensioni uguali a quello dei record.

Gli array di record e di indici

La lista dei nominativi inseriti è visibile in una finestra, che ha come titolo il nome del file in cui sono registrati i dati, che può essere spostata e le cui dimensioni possono essere modificate. La finestra è provvista di barre di scorrimento, sia orizzontali che verticali, che permettono di vedere le parti dell'elenco all'esterno della finestra.

La finestra principale

Con un click del mouse all'interno di questa finestra si seleziona un record (visualizzato in reverse) e con un doppio click si apre una finestra di dialogo (dialog) con cui si possono modificare i campi.

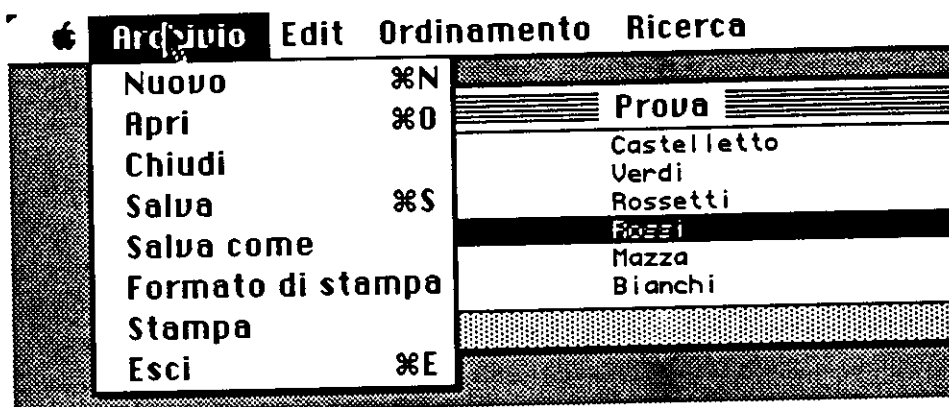
Lo stesso dialog viene utilizzato per inserire nuovi record. Sempre da questo dialog è possibile scorrere l'archivio, passando al record precedente o al successivo, e infine cancellare i record.

La finestra di dialogo

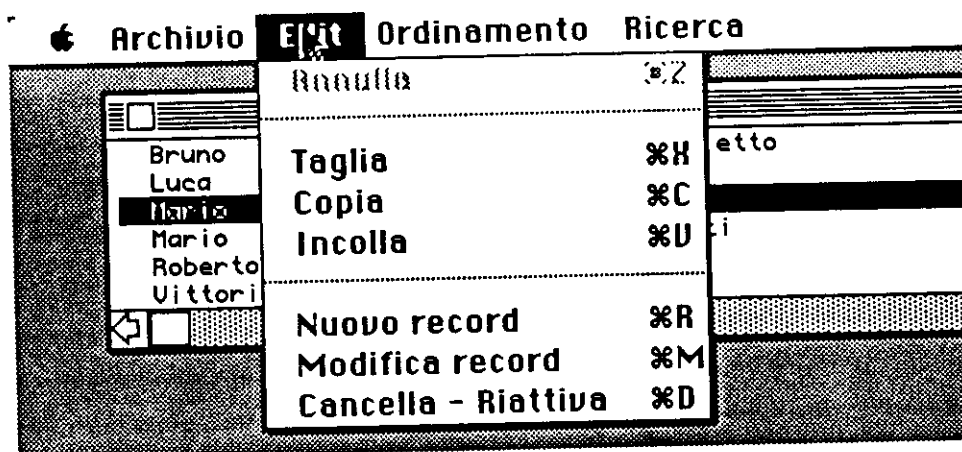
Un altro dialog che viene utilizzato è quello per la ricerca, con cui è possibile cercare il record successivo a quell corrente il cui campo chiave (quello secondo cui è selezionato l'ordinamento) contenga o inizi con una certa stringa. Il record (se trovato) sarà visualizzato nella finestra principale in reverse.

Il dialog di ricerca

La selezione del tipo di ordinamento avviene per mezzo dell'apposito menù.

Il menù per la selezione dell'ordinamento

Una volta selezionato il tipo di ordinamento, la finestra viene aggiornata e i record verranno visualizzati secondo l'ordine prescelto, le ricerche faranno riferimento al campo su cui sono ordinati i record e anche tutti i movimenti rispecchieranno questo ordinamento.

Le operazioni di archiviazione sono tutte governate dal menù **Archivio.** Con la selezione dell'opzione **Nuovo** si crea un nuovo indirizzario che viene visualizzato in una finestra dal titolo **Senza nome**, **Apri** permette di aprire archivi precedentemente creati, **Chiudi** chiude e salva il documento corrente, **Salva** lo registra e continua, **Salva come** permette di registrarlo con un nome differente da quello corrente, **Formato di stampa** e **Stampa** gestiscono le operazioni di stampa.

Il menù per le operazioni di archivio

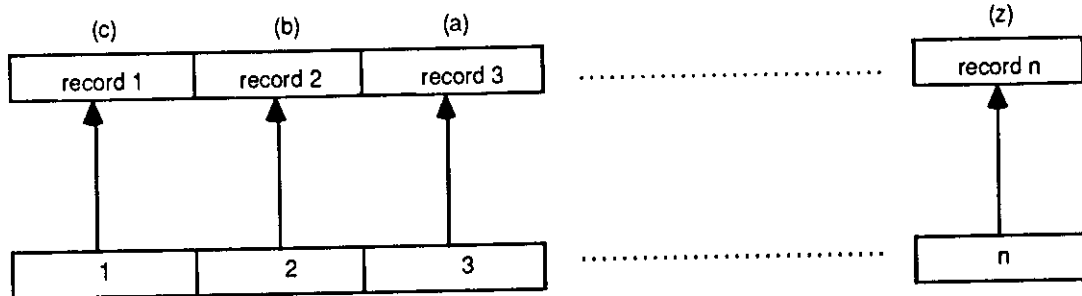
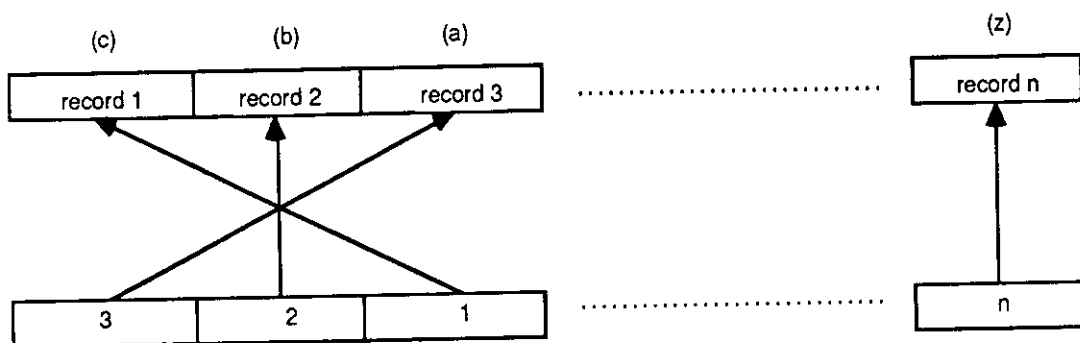
Le operazioni di modifica sono gestite dal menù **Edit**. Le operazioni di taglia copia e incolla seguono lo standard Macintosh e quindi permettono lo scambio di informazioni tra le applicazioni. I record che vengono tagliati o copiati vengono memorizzati nella memoria di comunicazione in formato Text con i vari campi separati da CR, mentre per incollare si possono utilizzare parti di testo in cui i campi siano separati da CR o TAB.

Il menù per le operazioni di modifica

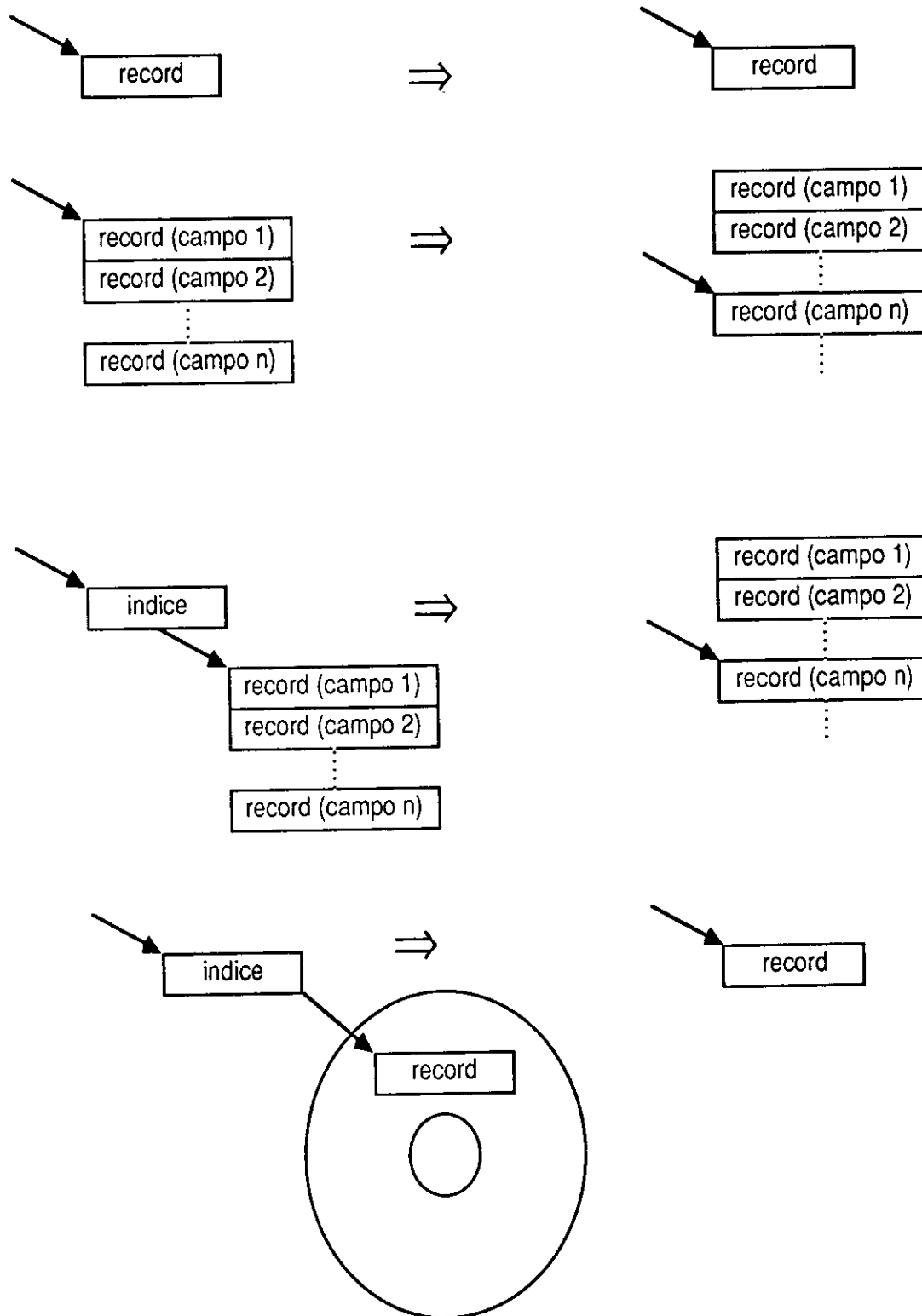
Le altre opzioni permettono di inserire nuovi record o modificare quello selezionato.

Finora abbiamo descritto a grandi linee le caratteristiche del programma per come lo vede l'utente, ora invece daremo uno sguardo all'organizzazione interna, cioè all'implementazione.

Come quasi tutti i programmi Mac anche il nostro è organizzato in modo event driven, cioè guidato dagli eventi. Ciò significa che il nucleo del programma è un ciclo, che si ripete fino a che non venga richiesta l'uscita dall'applicazione, che preleva da una coda gli eventi che gli vengono segnalati dal sistema operativo e in base al loro tipo invoca le procedure opportune. La maggior parte di queste riguardano l'uso del Toolbox e verranno esaminate nel capitolo relativo, mentre per quanto concerne l'applicazione vera e propria le parti più significative sono quelle che riguardano la gestione dei files e gli ordinamenti.

Per quanto riguarda la gestione dei files è stata scelta una soluzione molto semplice. I record del data base vengono scritti sui files nello stesso ordine in cui sono stati inseriti; quando il documento viene aperto tutti i record vengono caricati in memoria in un unico blocco e quindi vengono riordinati sul campo chiave scelto. I record cancellati durante una sessione di lavoro vengono semplicemente marcati come tali in memoria e poi solo al momento del salvataggio su disco si procede a scrivere solo quelli non cancellati.

Per l'ordinamento dei record viene utilizzato un insieme di procedure di sort generalizzato che usano il noto algoritmo di quick-sort. Abbiamo già visto in precedenza l'organizzazione interna dei dati (array di record e array di indici), vediamo ora nel dettaglio come avviene l'ordinamento.
L'ordinamento viene fatto sugli indici senza modificare la posizione dei record, in questo modo si evita lo spostamento dei numerosi bytes che compongono i record. Dopo il sort si potranno vedere i record ordinati seguendo la sequenza indicata nell'array di indici. Inizialmente l'array di indici conterrà i numeri in sequenza da 1 fino alla numero di record. Supponendo che il campo chiave sia di un carattere e che l'ordinamento sia di tipo alfabetico, vediamo in figura che i primi tre record sono fuori posto. Nella figura successiva vediamo invece come vengono riorganizzati gli indici dalla procedura di sort.

Situazione prima del sort

Situazione dopo il sort

La procedura di sort utilizzata richiede come parametri l'array da ordinare, la dimensione dei record e due funzioni. La prima è la funzione di comparazione che deve essere definita dal programma e permette quindi di utilizzare la stessa procedura di sort per ordinamenti di tipo numerico, alfabetico o di altro genere. La seconda è la funzione di accesso, quella che permette di estrarre dall'elemento dell'array da ordinare il campo chiave (nel nostro caso a partire dall'indice prende il record corrispondente e in questo il campo utilizzato per l'ordinamento). Combinando opportunamente le due funzioni è così possibile risolvere la maggior parte delle operazioni di ordinamento utilizzando sempre la stessa routine di base. Si può andare da semplice ordinamento diretto dell'array utilizzando l'identità, alla creazione di indici come nel nostro esempio, fino all' ordinamento di file su disco includendo una fase di lettura da disco.

Esempi di utilizzo della funzione di accesso

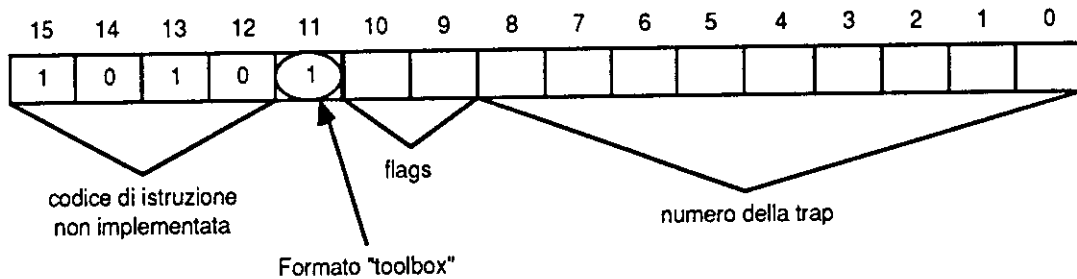
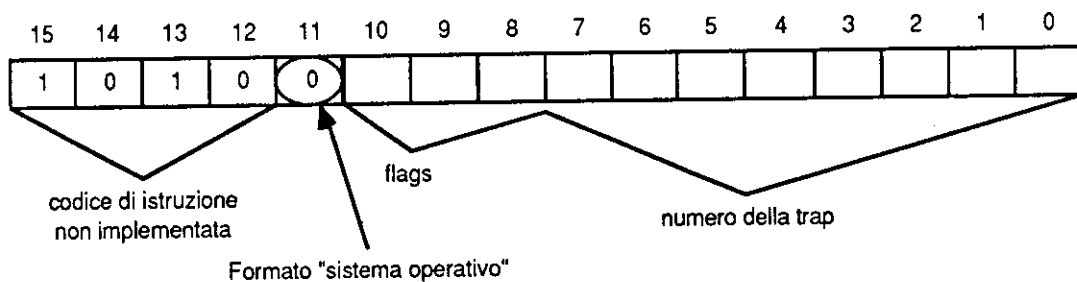
4 II Toolbox

Con il nome di **Toolbox** ci si riferisce a quell'insieme di routines di vario genere che risiedono nelle ROM di Macintosh e che costituiscono il cuore del sofisticato sistema di interfaccia.

Tutte le routine del Toolbox sono state sviluppate su Lisa utilizzando il Pascal e l'Assembler di quella macchina. Questo fatto non è solo di importanza storica, ma ha enormi conseguenze pratiche. Infatti tutte le chiamate al Toolbox rispettano le convenzioni e i formati di dati del Lisa Pascal. In un certo senso questo è il linguaggio nativo del Toolbox. In tutto il resto del documento la descrizione delle routines e le strutture dati saranno date in forma Pascal.

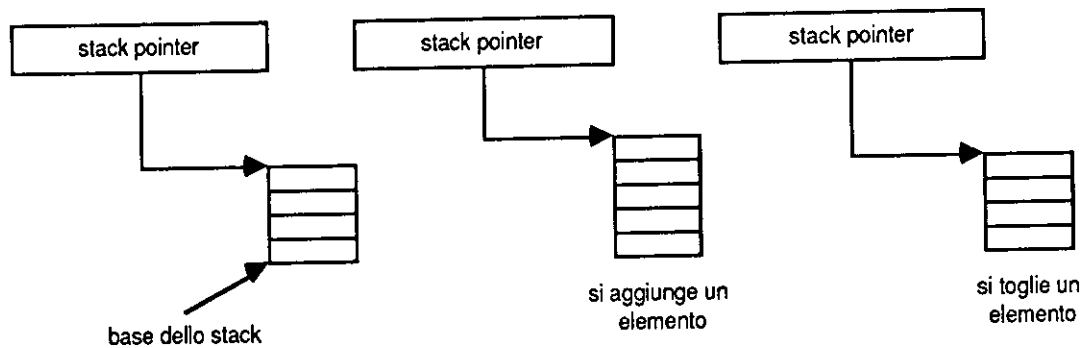
Il meccanismo di Trap.

A livello macchina tutte le chiamate al Toolbox devono essere tradotte in salti a subroutine in ROM. Il metodo con cui ciò viene realizzato in Mac è basato su una particolare caratteristica del 68000 detta "emulator trap" che permette di aggiungere nuove primitive al set di istruzioni. Queste istruzioni sembrano normali istruzioni del processore, ma non vengono eseguite direttamente, bensì emulate in software.

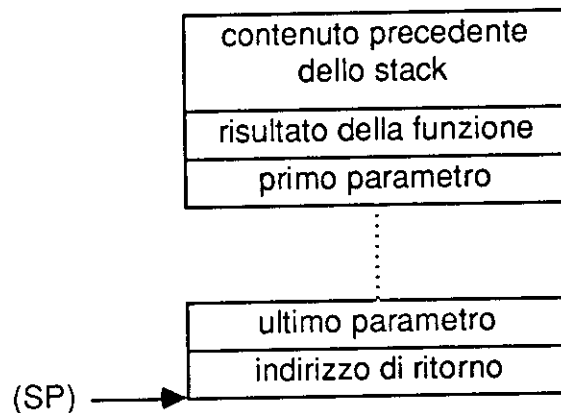
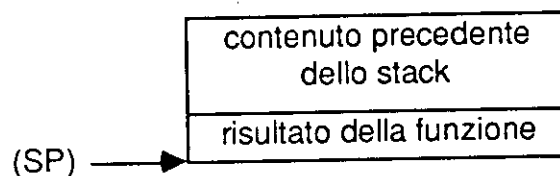
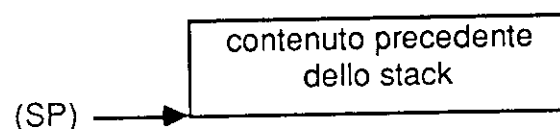
Una trap viene eseguita quando il processore verifica una anomalia nella esecuzione di un programma. In questo caso il processore interrompe l'esecuzione, salva il program counter e i registri e quindi esegue la routine di gestione della trap. Alla fine la routine di gestione della trap riporta i registri nelle condizioni iniziali e riprende l'esecuzione del programma interrotto. Le trap possono avvenire in diverse situazioni, come ad esempio una divisione per zero o un interrupt di I/O. Ogni tipo di trap ha la sua routine di gestione. In particolare la emulator trap si verifica quando il processore incontra una istruzione che non riconosce. Sul Macintosh viene utilizzato proprio questo meccanismo per eseguire le routine in ROM. In particolare per ogni routine del Toolbox viene utilizzata una parola i cui primi quattro bit sono **1010** e i seguenti dodici identificano la specifica chiamata.

Il formato "Toolbox" della parola di trap

Il formato "sistema operativo" della parola di trap

Una "istruzione" di questo tipo non viene riconosciuta come valida e il 68000 esegue la routine di trap (detta *trap dispatcher*) che a sua volta esamina l'istruzione e in base alla sua forma salta in ROM ad eseguire il codice opportuno.

Lo stack

Le routine scritte in Pascal ricevono i loro parametri e restituiscono i risultati sullo *stack*. Per capire come funziona lo stack si deve pensare a una pila di fogli. La base rimane sempre la stessa, ogni foglio viene aggiunto sempre in cima e quindi ogni volta che si prende un foglio dallo "stack", si prende l'ultimo che è stato aggiunto; lo stack cresce perciò in modo LIFO (Last In First Out).

Il comportamento dello stack

Anche le subroutines di un programma si comportano in modo LIFO, l'ultima routine chiamata è sempre la prima a ritornare al chiamante. Questo significa che i suoi parametri e la memoria privata possono essere tenuti in una memoria a stack. La base dello stack rimane fissa, mentre gli elementi vengono aggiunti o rimossi dalla testa dello stack, la cui posizione viene memorizzata in un registro detto *stack pointer*.

Quando si chiama una routine in Pascal (o in ogni linguaggio che segue le stesse convenzioni di chiamata), il compilatore genera una serie di istruzioni per aggiungere sulla cima dello stack i valori dei parametri, assieme all'indirizzo di ritorno (l'indirizzo dell'istruzione da eseguire una volta completata la routine). Se la routine é una funzione, allora viene anche riservato dello spazio sullo stack per il valore che deve restituire. La routine può poi eventualmente allocare spazio addizionale sullo stack per le proprie variabili locali. Se questa routine a sua volta ne chiama un'altra, lo spazio per i parametri e le variabili locali di quest'ultima verrà aggiunto sulla cima dello stack sopra quello della routine chiamante. Prima di restituire il controllo al punto in cui è stata chiamata, ogni routine toglie i suoi parametri, le variabili locali e l'indirizzo di ritorno dallo stack, lasciandolo così nello stesso stato in cui era prima della chiamata.

Lo stato dello stack alla chiamata di una funzione

Lo stato dello stack all'uscita di una funzione

Lo stato dello stack all'uscita di una procedura

Le routine del Toolbox sono per la maggior parte stack-based, cioé si aspettano i parametri sullo stack, mentre alcune sono register-based, cioé si aspettano i parametri nei registri. Nel primo caso il compilatore Pascal genera direttamente il codice per aggiungere i parametri sullo stack e la chiamata alla trap, mentre nel secondo caso in Pascal non si chiama direttamente la trap ma una speciale routine di interaccia che preleva gli argomenti dallo stack,dove vengono lasciati dal codice generato dal compilatore, e li carica nei registri prima di eseguire la trap.

Per chi usa linguaggi differenti dal Lisa Pascal è molto importante conoscere le convenzioni di passaggio di parametri di questo

linguaggio, dato che le routines del Toolbox si aspettano i parametri proprio in questa forma.

- Gli integers occupano 2 bytes, i long integers 4 bytes; entrambi in complemento a due.

- Tutti i puntatori occupano 4 bytes.

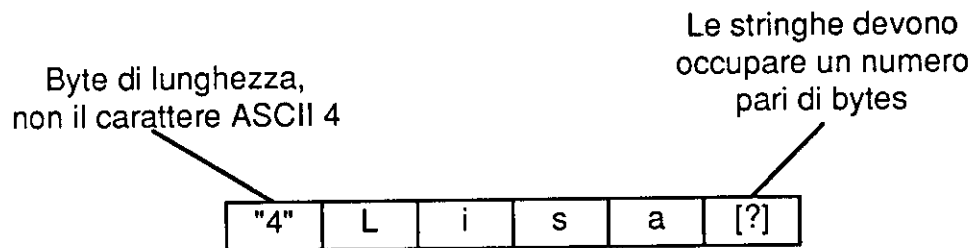
- I booleans occupano 2 bytes sullo stack, con il valore effettivo nel bit 8, il bit meno significativo del primo byte: **1 = TRUE, 0 = FALSE**. Gli altri 15 bit vengono ignorati.

- I singoli caratteri (quelli definiti come char in Pascal) occupano 2 bytes, con il codice ASCII nel secondo byte. Il primo byte é ignorato.

- Le stringhe di caratteri vengono rappresentate sullo stack da un puntatore (4 bytes) alla stringa reale in memoria. Il formato delle stringhe verrà descritto in seguito.

- Le strutture dati come records e array sono normalmente rappresentate con un puntatore (4 bytes) alla reale struttura in memoria. Se però la struttura occupa 4 bytes o meno, allora viene direttamente messa sullo stack al posto dell'indirizzo.

- Tutti i parametri variabili (quelli dichiarati come VAR in Pascal), indipendentemente dal tipo, vengono rappresentati con il puntatore alla variabile in memoria.

La routine rimuoverà i suoi parametri dallo stack prima di restituire il controllo al chiamante. Se la routine è una funzione, allora si deve riservare spazio sullo stack per il risultato prima di aggiungere i parametri; al ritorno dalla trap il risultato sarà sulla cima dello stack.
Il Toolbox utilizza per le stringhe di caratteri lo stesso formato del Lisa Pascal. Una stringa viene rappresentata internamente come una struttura a lunghezza variabile il cui primo byte rappresenta la lunghezza della stringa in caratteri. Poichè il contatore di caratteri è un byte la stringa può essere lunga al massimo 255 caratteri.

La rappresentazione interna delle stringhe

Le stringhe in questa forma vengono definite come tipo Pascal STR255 e vengono usate ad esempio per i titoli delle finestre e i menù. La stringa deve comunque occupare un numero pari di bytes; se il numero di bytes necessari dovesse essere dispari come in figura, viene aggiunto un byte inutilizzato (detto di "padding") alla fine; la stringa vuota perciò occuperà 2 bytes in memoria: un byte a 0 per il contatore e un byte di padding.

Il Finder e le applicazioni

Gli utenti di Mac, quando non utilizzano una applicazione, interagiscono con il Finder, che è quel particolare programma che gestisce l'interfaccia con il sistema operativo. Anche il Finder, come gli altri programmi per Mac, ha le sue peculiarità rispetto alle tradizionali "shell" di sistema operativo. Innanzitutto presenta le caratteristiche proprie dell'ambiente Mac, come finestre, controls, icone e uso del mouse, ma soprattutto cambia rispetto agli ambienti tradizionali il modo di "lanciare" programmi. Con il Finder non si lanciano i programmi, eventualmente specificando su quali dati questi devono operare, ma si possono "aprire" le applicazioni o anche i documenti (dati di una applicazione).

Il procedimento seguito dal finder quando si fa un doppio click su un file oppure si seleziona un file e si sceglie **Apri** dal menù **Archivio** è il seguente:

- se il file selezionato contiene un programma, allora il Finder lo lancia.
- se il file contiene un documento associato a qualche programma, allora il Finder lo lancia e gli indica di utilizzare i dati di quel documento.
- se il file non viene riconosciuto come associato a un programma o il programma non é presente sul disco, allora il Finder segnala con un alert box che non riesce a trovare una applicazione adatta.

Il Finder decide che cosa fare guardando due informazioni speciali che sono associate ad ogni file sul disco, il *tipo* e il *creatore* del file. Sono entrambi stringhe di quattro caratteri che devono essere specificate da ogni applicazione ogni volta che crea un nuovo file.

Il creatore associato ad ogni file indica al Finder quale programma lo ha creato, così è possibile lanciare l'applicazione quando si apre un documento. Il tipo invece indica il formato del documento, per cui si può segnalare all'applicazione che viene lanciata in quale formato sono i dati. Esistono due tipi standard di

interesse particolare. Un file che contiene un programma deve avere tipo **APPL** , mentre il tipo **TEXT** indica file di dati che contiene una sequenza di caratteri senza addizionali informazioni di formattazione o di altro genere.

Quando l'utente seleziona e apre uno o più documenti, il Finder guarda il creatore di questi e lancia quel file di tipo **APPL** che ha lo stesso creatore e gli passa come parametro una tabella di informazioni che identificano i documenti selezionati. Sarà poi compito dell'applicazione esaminare queste informazioni e aprire i file che contengono i dati.

Vediamo ora come nel nostro programma di esempio è stata realizzata la fase di inizializzazione e come viene gestita l'interazione con il Finder.

La procedura **initialize** è la prima ad essere chiamata dal programma principale e si occupa di inizializzare le variabili globali e di sistema per mezzo di opportune routines del Toolbox e della procedura **initvars**. In particolare all'inizio di ogni applicazione è necessario inizializzare QuickDraw per mezzo di *Initgraf*, cui si passa l'indirizzo di ThePort (una variabile globale di QuickDraw); poi si devono inizializzare il font manager e il window manager rispettivamente con *Initfonts* e *Initwindows*. E' buona norma che all'inizio di un'applicazione si chiami anche *FlushEvents* per eliminare ogni tipo di evento che può essere stato generato mentre l'applicazione veniva caricata. E' poi necessario inizializzare anche il menu manager con *Initmenus* (che viene chiamata all'interno di setupmenus), il TextEdit manager con *TEInit* e il Dialog manager con *Initdialogs* cui si può passare il puntatore ad una routine da eseguire in caso di errore.

```
PROCEDURE initialize;
 BEGIN
  initgraf(@thePort);
  initfonts;
  flushevents(everyevent, 0);
  setcursor(arrow);
  initwindows;
  setupmenus;
  teinit;
  initdialogs(NIL);
```

```
initvars;

{ Controllo se l'applicazione e' stata lanciata a
  partire da un documento }
countappfiles(mesg, filecnt);
IF filecnt > 0
 THEN
  BEGIN
   i := 1;
   REPEAT
    getappfiles(i, thefile);
    i := i + 1;
    { Apro solo files di tipo ARCH }
   UNTIL (i > filecnt) OR (thefile.ftype = 'ARCH');
   IF thefile.ftype <> 'ARCH'
    THEN
   { Nessuno dei file passati all'applicazione era di
     tipo ARCH }
     donew
    ELSE

     { Se è un file di tipo ARCH assegno ai campi del
       record globale Reply i valori opportuni e quindi
       chiamo la DoOpen per aprire il documento }
     WITH reply DO
      BEGIN
       Good := TRUE;
       fName := thefile.fName;
       vRefNum := thefile.vRefNum;
       doopen;
      END;
  END
 ELSE donew;
END;
```

Il resto della procedura è dedicato al riconoscimento dei files passati dal Finder all'applicazione. *CountAppFiles* restituisce nel secondo parametro il numero di files che sono stati selezionati e aperti dal Finder. Quindi con un ciclo di *GetAppFiles* si controlla il tipo di questi files (la nostra applicazione crea ed è in grado di leggere solo file di tipo ARCH) e si apre il primo file di tipo corretto che si incontra, altrimenti si apre un nuovo documento vuoto.

Gli eventi

Uno dei punti fermi dell'interfaccia Macintosh è che l'utente deve dire al computer che cosa deve fare e non viceversa. Troppo spesso con i sistemi tradizionali si ha la sensazione che usare un computer sia un esercizio di adattamento alla macchina, e che sia il computer a dare gli ordini:

```
Rispondere S o N
```

oppure

```
Premi un tasto per continuare
```

Se a questo punto non si replica con la risposta corretta si ottiene quasi sempre un messaggio di errore poco chiaro.

Macintosh è invece basato sul principio che il controllo deve essere in mano all'utente e non alla macchina. Invece di dare istruzioni su che cosa dovrà accadere in seguito, un programma nello stile Mac accetta istruzioni dall'utente riguardo a come proseguire. L'utente controlla il comportamento del programma con il mouse o con la tastiera; ogni azione di questo tipo costituisce un **evento** a cui il programma deve rispondere. Il programma spende la maggior parte del suo tempo in ozio, in attesa che accadano degli eventi, quindi risponde e attende il prossimo.

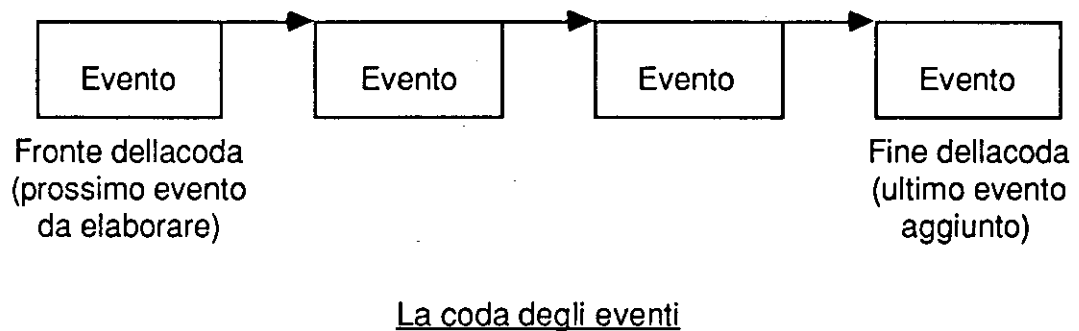
Un programma così organizzato viene detto *event-driven* o *guidato dagli eventi*. Il programma di esempio che utilizzeremo segue questa filosofia. Vediamo di seguito la struttura del programma principale.

```
PROGRAM sample;

 BEGIN          {* MAIN PROGRAM *}
  initialize;
  REPEAT
   systemtask;
   eventdptch;
  UNTIL doneflag;
  disposeall;
 END.
```

Come si può vedere il programma inizia con una chiamata alla procedura **initialize**, che abbiamo già visto in precedenza, e quindi entra nel cosidetto *main event loop* in cui ripetutamente viene chiamata la procedura di sistema *SystemTask* e la procedura **eventdptch** (Event Dispatcher) finchè il valore della variabile logica DoneFlag non sarà divenuto Vero. La chiamata a *SystemTask* è necessaria per la convivenza con gli accessori di scrivania; essa infatti toglie il controllo del processore al programma in esecuzione e lo cede agli accessori di scrivania che così possono eseguire le loro operazioni "contemporaneamente" al programma.
Eventdptch é il vero nucleo del programma event driven: controlla se si è verificato un evento e in tal caso a seconda del tipo e di altre caratteristiche invoca l'insieme di procedure che risponderanno all'evento stesso.

Al livello logico più basso gli eventi vengono individuati e registrati con un meccanismo a interrupt. Quando l'utente preme il bottone del mouse o un tasto, viene inviato a Macintosh un segnale elettrico che fa sì che il processore sospenda temporaneamente quello che stava facendo e esegua subito una speciale routine di *gestione interrupt.* La routine non risponde direttamente all'evento, ma memorizza le circostanze in cui è avvenuto (quando è avvenuto, dove era il mouse, quale tasto è stato premuto ecc.) per una futura elaborazione. Quindi restituisce il controllo al programma in esecuzione, che prosegue come se nulla fosse accaduto. In seguito, quando il programma sarà pronto a elaborare l'evento, potrà recuperare le informazioni memorizzate e eseguire le operazioni necessarie.

La lista in cui gli eventi vengono memorizzati per la susseguente elaborazione viene chiamata *event queue*. Ogni elemento in questa coda è un *event record* che rappresenta un singolo evento.

La coda degli eventi

Ci sono sedici possibili tipi di eventi, identificati da codici da 0 a 15. I più importanti cadono nelle seguenti categorie.

- *Mouse events*: l'utente ha premuto il bottone del mouse (evento *mouseDown*) o lo ha rilasciato (evento *mouseUp*).
- *Keyboard events*: l'utente ha premuto un tasto della tastiera (evento *keyDown*), lo ha tenuto premuto finché non ha iniziato a ripetere automaticamente (evento *autoKey*) o lo ha rilasciato (evento *keyUp*).
- *Disk-inserted events*: l'utente ha inserito un disco in un drive.
- *Window events*: l'utente ha attivato o disattivato una finestra (evento *activateEvt*), oppure ha esposto una parte di essa precedentemente nascosta (evento *updateEvt*)
- *Null events*: il programma ha chiesto al toolbox riguardo un evento ma non ce ne era nessuno in coda.

Tutto quello che un programma ha bisogno di sapere riguardo un evento è contenuto in un event record la cui definizione in Pascal è la seguente:

```
EventRecord = record
  what       : INTEGER; {tipo di evento}
  message    : LONGINT; {informazioni dipendenti dal tipo}
  when       : LONGINT; {istante in cui é avvenuto}
  where      : Point;   {posizione del mouse}
  modifiers  : INTEGER; {stato dei tasti di modifica}
end;
```

Analizzando le informazioni contenute in un event record il

programma può determinare cosa significa quell'evento e come rispondere. L'informazione più importante è contenuta nel campo **what**, che indica il tipo di evento. Il campo **message** contiene un messaggio che dà ulteriori informazioni sull'evento. Ogni tipo di evento usa questo campo con un significato particolare; ad esempio per i window events in questo campo c'è il puntatore alla finestra relativa all'evento, per i keyboard events contiene sia il codice del tasto che il codice ASCII del carattere corrispondente al tasto premuto o rilasciato.
Il resto dell'informazione nell'event record è indipendente dal tipo di evento. Il campo **when** indica l'istante (secondo l'orologio di sistema) in cui è avvenuto l'evento, il campo **where** contiene le coordinate del mouse sullo schermo e il campo **modifiers** indica lo stato dei tasti di modifica (maiuscole, option, command, bottone del mouse ecc.).

Normalmente si trattano gli eventi chiamando la funzione del Toolbox *GetNextEvent*. A questa funzione si passa una maschera che indica a quali eventi si è interessati. Il Toolbox trova nella coda il primo evento che si accorda alla maschera e lo restituisce.

La routine che costituisce il cuore del nostro programma event driven è **eventdptch**. Il suo compito, ogni volta che viene chiamata nel main loop, è di prelevare un evento dalla coda degli eventi e di far eseguire le operazioni opportune.

La procedura inizia con una chiamata a *GetNextEvent* con la maschera EveryEvent che richiede il primo evento di qualsiasi tipo. Nel caso fosse presente un evento nella coda lo esamineremo e a seconda del tipo (case analysis su myevent.what) andremo a recuperare altre informazioni. Se è stato premuto il bottone del mouse, chiameremo *FindWindow* per capire in che finestra e in quale parte di questa era il mouse in quel momento. Se invece è stato premuto un tasto (la nostra applicazione è interessata solo ai cosiddetti equivalenti di menù) si eseguirà il comando di menù equivalente. Se è un evento di tipo activate si guarderà il campo modifiers per capire se è una attivazione o una deattivazione della finestra. Se infine è un evento update, si provvederà a ridisegnare la parte di finestra da aggiornare.

```
PROCEDURE eventdptch;

  BEGIN
   IF GetNextEvent(everyevent, myevent)
    THEN
     CASE myevent.what OF

      mousedown:
       BEGIN
        code := findwindow(myevent.where, whichwindow);
        CASE code OF

         inmenubar: docommand(menuselect(myevent.where));

         insyswindow: systemclick(myevent, whichwindow);

         indrag: dragwindow(whichwindow, myevent.where,
                            dragrect);

         ingoaway:
          IF trackgoaway(whichwindow, myevent.where)
           THEN doclose;

         ingrow:
          IF whichwindow = frontwindow
           THEN growwnd(whichwindow)
           ELSE selectwindow(whichwindow);

         incontent:
          BEGIN
           IF whichwindow <> frontwindow
            THEN selectwindow(whichwindow)
            ELSE
             BEGIN          { è la finestra attiva }
              GlobalToLocal(myevent.where);
              { e' nella parte di testo ? }
              IF ptinrect(myevent.where, prect)
               THEN clickincont(myevent)
               ELSE
                BEGIN                {è nei controls}
                 controlzone := findcontrol(myevent.where,
                              whichwindow, whichcontrol);
                 CASE controlzone OF
                  inUpButton:
                   t := trackcontrol(whichcontrol,
                             myevent.where, @scrollup);
                  inDownButton:
                   t := trackcontrol(whichcontrol,
                           myevent.where, @scrolldown);
                  inpageup: pagescroll(controlzone, - 10);
                  inpagedown: pagescroll(controlzone, 10);
```

```
                inthumb:
                 BEGIN
                  t := trackcontrol(whichcontrol,
                                    myevent.where, NIL);
                  scrollbits
                 END
               END           {* Case ControlZone *}
              END                     {* controls *}
           END                        {* front *}
        END                           {* in Content *}
      END                             {* of code case *}
     END;                             {* of mouseDown *}

    keydown, autokey:
     BEGIN
      IF thewindow = frontwindow
       THEN
        BEGIN
         thechar := CHR(myevent.message MOD 256);
         { se il tasto Command e' premuto }
         IF bitand(myevent.modifiers, 256) <> 0
          THEN docommand(menukey(thechar));
        END
     END;               {* of keyDown *}

    activateevt:
     BEGIN
      DrawGrowIcon(thewindow);
      IF odd(myevent.modifiers)
       THEN             { la window diventa attiva }
        BEGIN
         SetPort(thewindow);
         teactivate(hte);
         ShowControl(vscroll);
         ShowControl(hscroll)
        END
       ELSE            { la window viene disattivata }
        BEGIN
         tedeactivate(hte);
         HideControl(vscroll);
         HideControl(hscroll)
        END
     END;               {* of activateEvt *}

    updateevt:
     BEGIN
      GetPort(saveport);
      SetPort(thewindow);
      beginupdate(thewindow);
      drawwindow(thewindow);
      endupdate(thewindow);
```

```
        SetPort(saveport);
      END                   {* of updateEvt *}

    END                     {* of event case *}
  END;                      {* EventDptch *}
```

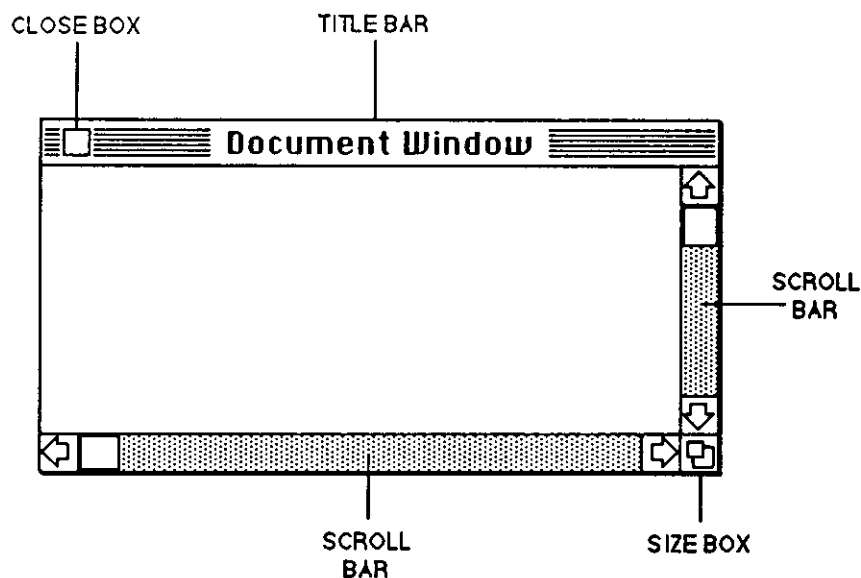
Ora analizziamo come vengono trattati i click avvenuti nella parte della finestra che contiene il testo. Questo compito viene affidato alla procedura **ClickInCont** cui si passa l' eventRecord rilevato. La procedura determina a quale record corrisponde la riga su cui è stato premuto il mouse, in base al punto (prEvent.where), all'altezza dei caratteri utilizzati (hte^.lineHeight) e tenendo presente di quanto il testo è stato spostato con le scroll bars in precedenza (theOrigin.v); seleziona poi il record in questione chiamando **SelTheLine** e rimane in attesa per mezzo secondo; quindi controlla se c'è stato un secondo click, chiamando *GetNextEvent* e richiedendo solo gli eventi di tipo mouseDown. La funzione **biclick** ci dice se questo secondo click è avvenuto nella stessa zona del primo e entro un tempo tale da giudicarlo un doppio click; in tal caso si apre un dialog per la modifica dei dati del record.

```
PROCEDURE clickincont(prEvent: EventRecord);

 VAR
  secevent  : EventRecord;
  i         : INTEGER;

 FUNCTION biclick(uno, due: EventRecord): Boolean;

  BEGIN
   biclick := (abs(uno.where.h - due.where.h) < 5) AND
              (abs(uno.where.v - due.where.v) < 5) AND
              (due.when - uno.when < 30);
  END;

 BEGIN       {ClickinCont}

  currecind := (prEvent.where.v + theorigin.v) DIV
               hte^^.lineHeight;
  IF currecind > numrec - 1
   THEN currecind := numrec - 1;
  seltheline(currecind);
  Delay(30, ticks); { Attende 1/2 secondo per dare tempo
                      di effettuare il secondo click }

  { Se c'e' un secondo click controlliamo (con biClick)
```

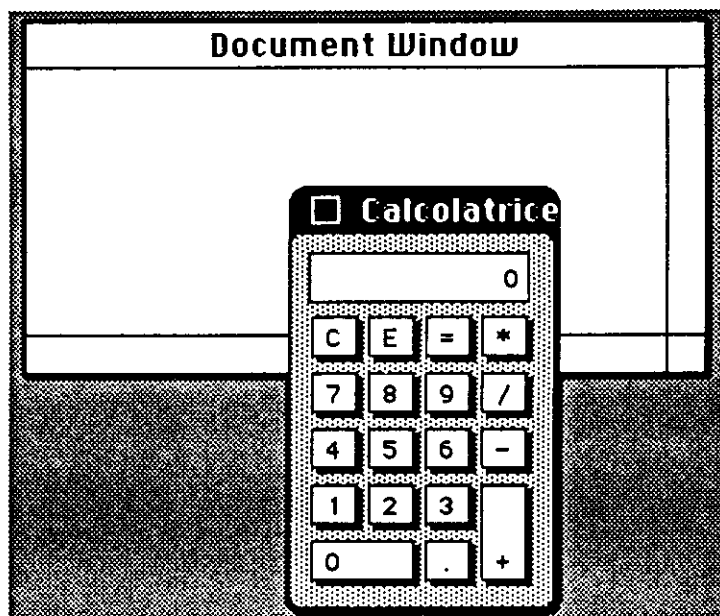
```
      se e' avvenuto nella stessa zona e se cosi' apriamo
      il dialog per modificare il record selezionato }
    IF GetNextEvent(MDownMask, secevent)
     THEN
      BEGIN
       GlobalToLocal(secevent.where);
       IF biclick(prEvent, secevent)
        THEN openrecord(theindhdl^^[currecind]);
      END;
  END;
```

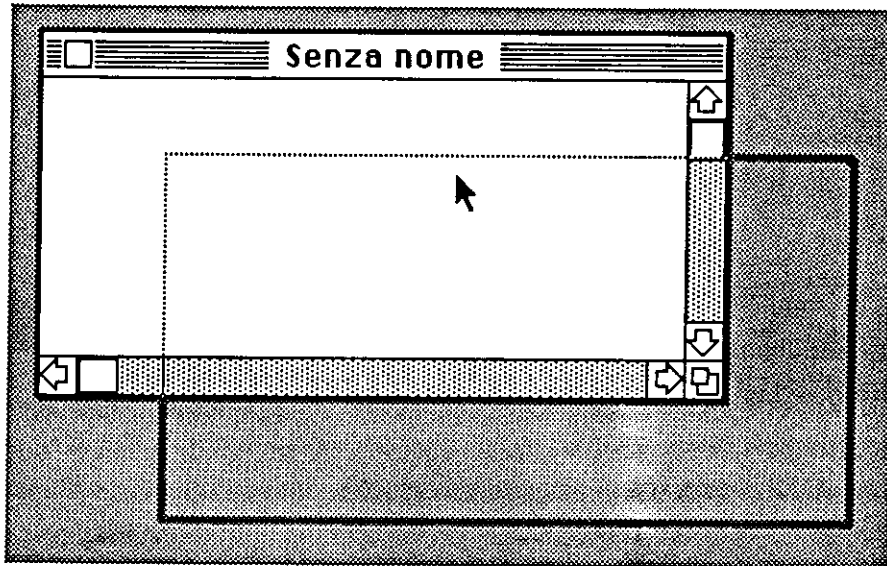
Le finestre

Le finestre sono un concetto noto a chiunque abbia usato anche poco Macintosh. Dal punto di vista del programma si possono pensare come un canale di comunicazione bidirezionale con l'utente: il programma fornisce informazioni all'utente mostrandole nella finestra e l'utente dice al programma che cosa fare con dei click del mouse in punti strategici della finestra.
Ci sono diversi tipi di finestre, ognuna con le proprie caratteristiche, ma tutte si conformano nel comportamento agli standard definiti nella parte di *Inside Macintosh* dal titolo Macintosh User Interface Guidelines.

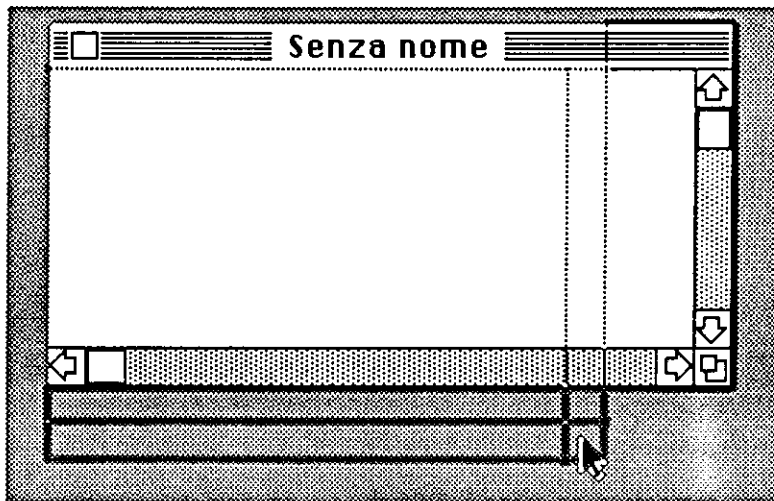
In linea di principio ci possono essere un numero illimitato di finestre sullo schermo e si possono sovrapporre in ogni ordine. Alcune finestre vengono create dal programma, mentre altre dal sistema (come quelle di alcuni accessori di scrivania).
Nella figura seguente vediamo la struttura standard di una finestra di tipo *document.* Nella parte più alta della finestra abbiamo la *title bar* che ne mostra il titolo. All'interno della title bar a sinistra c'è la *close box* . Nell'angolo in basso a destra si trova la *size box.* Una finestra può anche contenere dei *controls* di vario tipo, come in questo caso delle *scroll bars.*

Le parti di una finestra di tipo Document

Nelle già menzionate User Interface Guidelines si prescrivono dei comportamenti che le finestre devono seguire:

- quando la finestra è attiva la title bar deve essere evidenziata, mentre quando è inattiva l'evidenziazione deve essere eliminata. Un click in una parte qualsiasi di una finestra inattiva deve renderla attiva e portarla sopra tutte le altre.

Una finestra inattiva e un accessorio di scrivania

- premendo il pulsante del mouse nella title bar e spostandosi sullo schermo tenendolo premuto ("dragging"), si deve ottenere l'effetto di spostare il contorno della finestra, che segue il movimento del mouse. Quando il pulsante viene rilasciato la finestra viene ridisegnata nella nuova posizione raggiunta sullo schermo.

Lo spostamento di una finestra

- un click nella close box fa scomparire la finestra dallo schermo. Una finestra che non può essere chiusa dall'utente non deve quindi avere close box.

- quando l'utente preme il bottone del mouse nella size box e lo sposta, la sagoma della finestra viene disegnata con l'angolo alto a sinistra ancorato mentre l'estremo in basso a sinistra segue i movimenti del mouse finchè il pulsante non viene rilasciato. A questo punto la finestra viene ridisegnata con le nuove dimensioni.

Il ridimensionamento di una finestra

Tutti questi comportamenti delle finestre sono facilmente ottenibili, dato che il Toolbox mette a disposizione delle routines pronte ad essere usate per questo scopo.
Ritornando a vedere la procedura eventdptch mostrata nella parte riguardante gli eventi, possiamo ritrovarle:
nel caso di un evento mouseDown si chiama la funzione *FindWindow* passandole l'event record relativo al click del mouse; essa ritorna in un parametro var la finestra in cui il click è avvenuto e ha come valore il codice della parte della finestra interessata al click. A questo punto il gioco è quasi fatto:

- se il click è nella barra dei menù (inMenuBar) ne riparliamo più avanti

- se è nella finestra di sistema (inSysWindow) è presumibilmente un click in un accessorio di scrivania e quindi ripassiamo l'evento al sistema con *SystemClick*

- se è nella title bar (inDrag) chiamiamo la procedura *DragWindow* che provvede allo spostamento della finestra, passandole il rettangolo massimo in cui deve restare la finestra quando viene spostata

- se è nella close box (inGoAway) controlliamo con *TrackGoAway* che il mouse venga rilasciato mentre è ancora

nella close box (escludiamo perciò i click involontari e diamo all'utente la possibilità, spostando il mouse prima di rilasciarlo, di ripensarci). Infine chiudiamo la finestra

- se è nella size box (inGrow) e la finestra è già quella attiva, chiamiamo la procedura **growWnd**, che vedremo tra poco, per cambiare le dimensioni della finestra. Se la finestra era inattiva la rendiamo semplicemente attiva con *SelectWindow.*

- se è nel contenuto della finestra (inContent), ma la finestra è inattiva la attiviamo semplicemente;
  se invece la finestra è già quella attiva, guardiamo se il click è nella parte di testo (cioè il punto myevent.where è nel rettangolo pRect in cui è contenuto il testo) oppure, per esclusione, nei controls.
  Nel primo caso chiamiamo la nostra procedura **clickincont** per selezionare il record su quella linea o per modificarlo (se è un doppio click). Nel secondo caso si chiamano le procedure per lo scorrimento del testo che vedremo nel capitolo relativo ai controls.

```
PROCEDURE growwnd(whichwindow: WindowPtr);
{Ridimensiona la finestra e segnala le update region }

 VAR
  longresult: LONGINT;
  height,
  width: INTEGER; { Le nuove dimensioni della finestra }
  trect    : Rect; { Rettangolo da ridisegnare }

 BEGIN
  longresult := GrowWindow(whichwindow, myevent.where,
                           growrect);
  IF longresult = 0
   THEN EXIT(growwnd);
  height := hiword(longresult);
  width := loword(longresult);

  IF height < 64
   THEN height := 64;
  IF width < 64
   THEN width := 64;

  {aggiunge la vecchia "scroll bar area" alla update
   region in modo che venga ridisegnta (se la finestra
```

```
  viene ingrandita)}
 trect := whichwindow^.portRect;
 trect.left := trect.right - 16;
 InvalRect(trect);
 trect := whichwindow^.portRect;
 trect.top := trect.bottom - 16;
 InvalRect(trect);

{ ridisegna la finestra con le nuove dimensioni}
 SizeWindow(whichwindow, width, height, TRUE);
 movescrollbars;
 resizeprect;

 hte^^.ViewRect := prect;{sistema viewRect per TextEdit}
 InvalRect(prect);

 {aggiunge la nuova "scroll bar area" alla update region
  in modo che venga ridisegnta (quando la finestra viene
  rimpicciolita)}
 trect := whichwindow^.portRect;
 trect.left := trect.right - 16;
 InvalRect(trect);
 trect := whichwindow^.portRect;
 trect.top := trect.bottom - 16;
 InvalRect(trect);
END;  {* of GrowWnd *}
```

La procedura **growwnd** inizia con una chiamata alla funzione *GrowWindow*, che segue i movimenti del mouse e disegna la sagoma della finestra cui si cambiano le dimensioni, e restituisce in un LONGINT le nuove dimensioni della finestra: nella parte alta (*Hiword*) l'altezza e in quella bassa (*Loword*) la larghezza.
Per evitare problemi di visualizzazione delle scroll bars diamo un minimo alle dimensioni della finestra (64 pixels), quindi segnaliamo al sistema che le zone precedentemente occupate dalle scroll bars dovranno essere ridisegnate (*Invalrect*), cambiamo effettivamente le dimensioni della finestra (*SizeWindow*), mettiamo le scroll bars nella giusta posizione rispetto alle nuove dimensioni della finestra (**MoveScrollBars**), aggiorniamo le dimensioni del rettangolo in cui si disegna il testo (**ResizepRect**) e infine segnaliamo al sistema che anche le zone in cui andranno le nuove scroll bars dovranno essere ridisegnate.

```
PROCEDURE resizeprect;

 BEGIN
  prect := thePort^.portRect;
  prect.left := prect.left + 4;
  prect.right := prect.right - 15;
  prect.bottom := prect.bottom - 15
 END;
```

Il compito di **ResizepRect** è semplicemente quello di portare prect (il rettangolo in cui si disegna il testo) alle dimensioni corrette rispetto a quelle che la finestra ha in quel momento, lasciando liberi a sinistra 4 pixels, per evitare la sgradevole sensazione di vedere il testo attaccato ad un lato della finestra, e 15 pixels in basso e a destra per le scroll bars.

Creazione e distruzione di finestre

Prima di poter usare qualsiasi finestra è necessario inizializzare QuickDraw con *InitGraf*, la gestione delle fonti con *InitFonts* e il Window Manager stesso con *InitWindows.* A questo punto si possono creare tutte le finestre necessarie con *NewWindow* o *GetNewWindow* e distruggerle con *DisposeWindow* o *CloseWindow. NewWindow* crea una nuova finestra, di cui bisogna passare come parametri tutte le caratteristiche, e restituisce come valore il WindowPtr relativo. *GetNewWindow* si comporta esattamente come *NewWindow,* ma invece di passare tutti i parametri basta l'indice di una risorsa in cui questi sono definiti.

```
PROCEDURE wincreate;

 BEGIN

  { Azzera le tabelle di indici e records. }
  SetHandleSize(Handle(thetabhdl), 0);
  SetHandleSize(Handle(theindhdl), 0);

  numrec := 0;
  currecind := 0;
  dirty := False;
  { Crea la window caricandola dalle risorse }
```

```
    thewindow := GetNewWindow(256, @wrecord, POINTER(-1));
    SetPort(thewindow);

    { Setta fonte e dimensione }
    thewindow^.txFont := MYTXFONT;
    thewindow^.txsize := MYTXSIZE;
    resizeprect;

    { Istanzia un record di editing }
    hte := TENew(prect, prect);
    winexist := TRUE;

    { TE andrà a capo solo coi carriage-return }
    hte^^.crOnly := - 1;

    { Carica le scroll bars dalle risorse }
    vscroll := GetNewControl(256, thewindow);
    hscroll := GetNewControl(257, thewindow);

    theorigin.h := 0;
    theorigin.v := 0;
  END;
```

La procedura **WinCreate** azzera lo spazio allocato per le tabelle dei record e alcune variabili globali, quindi chiama *GetNewWindow* con parametri 256, che è l'indice della risorsa in cui è definita la finestra, il puntatore a wrecord, che sarà il WindowRecord di questa finestra, e infine POINTER(-1), che indica che la finestra dovrà essere quella esposta sopra tutte le altre eventualmente presenti sullo schermo. In seguito si indica a QuickDraw di dirigere l'output grafico su questa finestra con *SetPort*, e si fissano la fonte e le dimensioni dei caratteri che vi appariranno. Poi vengono eseguite operazioni che analizzeremo in seguito, come creare un nuovo TextEdit record e associare le scroll bars alla finestra.

```
Type WIND
  ,256
Senza nome
  50 40 300 450
  Visible GoAway
  0
  0
```

Qui sopra vediamo la definizione della risorsa relativa alla finestra di indice 256, dal titolo "Senza nome", nel rettangolo di coordinate 50 40 300 450, visibile (Visible) e con una close box (GoAway), è

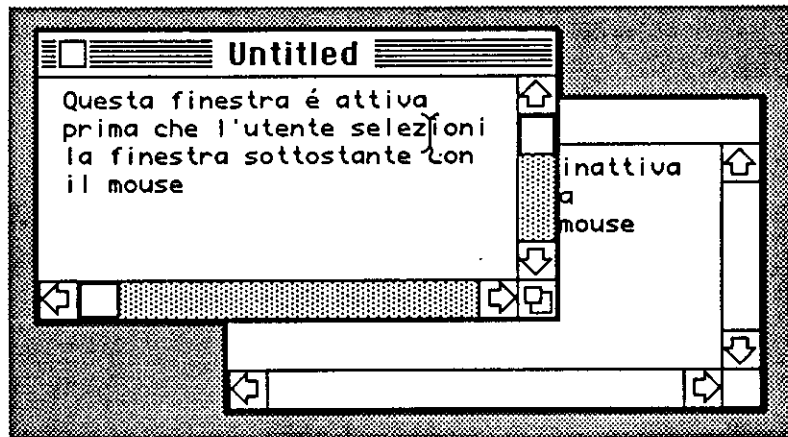
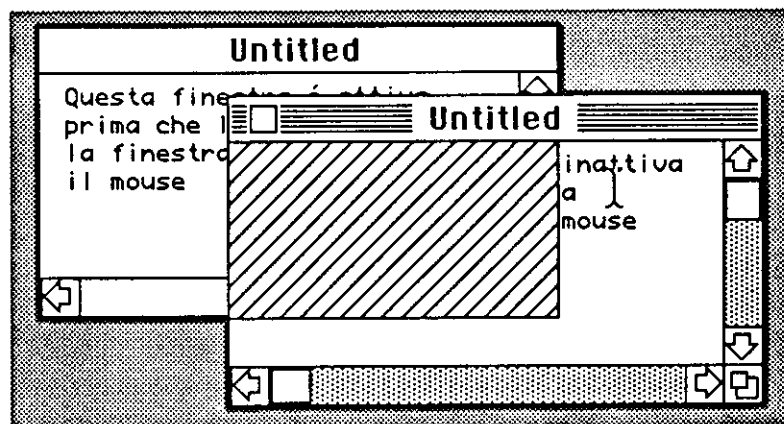
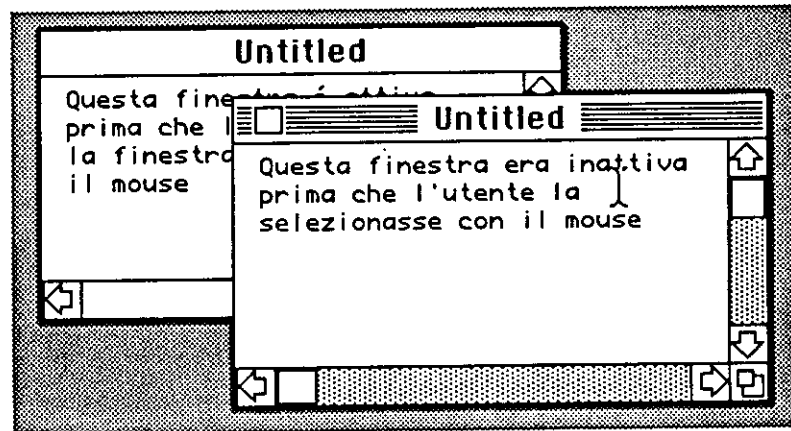
di tipo Document (0) e infine ha numero di riferimento 0.

Eventi relativi alle finestre

Quando l'utente manipola le finestre sullo schermo, il Toolbox ricorda le modifiche e le segnala al programma per mezzo di eventi relativi alle finestre. Ad esempio quando l'utente clicka su una finestra inattiva per renderla attiva il programma riceverà un evento *deactivate* per la finestra correntemente attiva e un *activate* per la finestra che deve diventare attiva. Quando una parte di una finestra che precedentemente era coperta diventa esposta il programma riceve un evento *update* che segnala di ridisegnare quella parte della finestra. Gli eventi relativi alle finestre non vengono registrati nella coda con il meccanismo di interrupt, ma possono essere comunque letti normalmente con GetNextEvent.

Gli eventi *activate* e *deactivate* avvengono normalmente in coppia: una finestra diventa attiva mentre un'altra diventa contemporaneamente inattiva. Entrambi questi eventi sono del tipo activate e si distinguono a seconda del valore del campo modifiers dell'event record, mentre nel campo message si trova il puntatore alla finestra interessata all'evento. Nella procedura eventdptch mostrata in precedenza possiamo osservare le azioni da compiere quando si rilevano degli eventi activate. Si chiama la procedura *DrawGrowIcon* per ridisegnare l'icona della size box coerentemente con lo stato della finestra (in una document window la size box è parte del contenuto della finestra e quindi la sua visualizzazione è a carico del programma). Quindi se la finestra diventa attiva (il bit 0 del campo modifiers è a 1), si deve attivare il testo e mostrare le scroll bars; se invece la finestra diventa inattiva, allora bisogna disattivare il testo e cancellare le scroll bars.

Tutte le volte che l'utente chiude, muove o ridimensiona una finestra, il Toolbox controlla quali parti di questa e di altre finestre diventano esposte e hanno quindi bisogno di essere ridisegnate sullo schermo. Il Toolbox di per sè ridisegna il contorno delle finestre, ma il contenuto è a carico del programma.
Gli eventi di tipo *update* hanno nel campo *message* il puntatore

alla finestra da modificare. Sempre in eventdptch possiamo vedere come il nostro programma tratta gli eventi di tipo update. Si salva la porta corrente in una variabile temporanea, si setta come porta corrente quella indicata nel campo message, si chiama *BeginUpdate*, si ridisegna la finestra chiamando **DrawWindow**, si chiama *EndUpdate* e infine si resetta come porta corrente quella che era in precedenza.

La situazione prima di attivare una finestra

Dopo la selezione (è evidenziata la regione di update)

La situazione dopo l'aggiornamento della regione di update

BeginUpdate salva una copia della regione visibile della finestra, quindi temporaneamente restringe le operazioni all'intersezione di questa con la regione di update. Per le normali finestre di tipo document non si deve dimenticare che la size box e le scroll bars fanno parte del contenuto della finestra e quindi devono essere incluse nelle operazioni di ridisegno. Finite le operazioni di aggiornamento, *EndUpdate* resetta la regione visibile della finestra ai suoi valori originari e annulla la regione di update.

```
PROCEDURE drawwindow(whichwindow: WindowPtr);
{Disegna il contenuto della finestra, dopo aver
 cancellato quanto conteneva prima}

 BEGIN
  {Il rettangolo di clipping è uguale al rettangolo della
   porta}
  ClipRect(whichwindow^.portRect);
  {Cancellalo}
  EraseRect(whichwindow^.portRect);
  {Disegna l'icona di crescita}
  DrawGrowIcon(whichwindow);
  {Disegna i controlli}
  DrawControls(whichwindow);
  {Aggiorna il rettangolo di testo}
  TEUpdate(prect, hte)
 END;
```

I menù

I menù pull-down di Macintosh seguono la filosofia generale dell'interfaccia Mac secondo cui spetta all'utente il controllo delle operazioni. I menù pull-down non occupano lo schermo richiedendo risposte e impedendo altre attività, ma attendono pazientemente, nascosti sotto la barra in alto nello schermo, che l'utente ne richieda la presenza con un click del mouse.

La maggior parte del tempo la parte visibile dei menù è ristretta alla barra nella parte superiore dello schermo, che elenca i titoli dei menù disponibili. Tenendo premuto il mouse su uno di questi l'utente fa apparire il menù vero e proprio, che mostra un lista di opzioni (*items*) fra cui scegliere. Spostando il mouse, sempre con il bottone premuto, viene evidenziato l'item che sta sotto il mouse; rilasciando il bottone si sceglie come opzione l'item che era evidenziato.

In figura possiamo vedere un tipico menù. Ogni item è composto da una linea di testo. Gli item che non possono essere scelti sono mostrati in grigio e si dice che sono disabilitati. Un programma può definire eventualmente un equivalente di tastiera per ogni item di menù, ovvero un carattere che, premuto contemporaneamente al tasto "command", abbia lo stesso effetto di un click in quell'*item*. Questi equivalenti, se ci sono, sono mostrati alla destra degli *items* relativi, preceduti dal simbolo corrispondente al tasto "command".

Composizione

Non posso annullare ⌘Z

Taglia ⌘X
Copia ⌘C
Incolla ⌘V

Mostra gli Appunti

Un menù standard

Il testo degli items di menù viene sempre mostrato nella fonte standard di sistema, ma è possibile modificarne lo stile (grassetto, sottolineato, ecc.). Inoltre si può anche marcare un item con un simbolo (*checkmark*); questo è utile quando l'item di menù serve a modificare lo stato di una opzione come un interruttore: la presenza o l'assenza del simbolo indica lo stato della opzione.

Stile	
✓Normale	⌘P
Grassetto	⌘B
Corsivo	⌘I
Sottolineato	⌘U
Bordato	⌘O
Ombreggiato	⌘S
✓Allinea a Sinistra	⌘L
Allinea al Centro	⌘M
Allinea a Destra	⌘R

Stile dei caratteri e simboli di check

Il primo passo da compiere nella costruzione dei menù e quello di definirli e di installarli nella barra dei menù. A questo fine si possono seguire due strade: la prima è quella di costruirli da zero, iniziando con un menù vuoto e aggiungendo items uno alla volta, la seconda è quella di leggere la definizione da una risorsa.

Nel nostro programma la creazione dei menù viene fatta nella procedura **SetUpMenus**. I menù devono essere definiti solo una volta, all'inizio del programma, per cui questa procedura viene chiamata nella parte di inizializzazione. Prima di tentare qualsiasi operazione con i menù bisogna inizializzare il menù manager con *InitMenus*, che deve essere preceduta dalle chiamate a *InitGraf*, *InitFonts* e *InitWindows*.

```
PROCEDURE setupmenus;  { inizializazione dei menus }

 VAR
  i        : INTEGER;

 BEGIN
  Initmenus;     { inizializza il menù Manager }
  {Carica nel primo elemento dell'array il menù }
      mymenus[1] := getmenu(1);
  addresmenu(mymenus[1], 'DRVR'); { desk accessories }
  mymenus[FILEID] := getmenu(256);
  mymenus[EDITID] := getmenu(257);
  mymenus[SORTID] := getmenu(258);
  mymenus[SEARCHID] := getmenu(259);
  FOR i := 1 TO LASTMENU DO
   insertmenu(mymenus[i], 0);
  DrawMenuBar;
 END; {* of SetUpMenus *}
```

La procedura chiama *GetMenu* per ogni menù passando l'indice con cui è stato definito nel file delle risorse e carica gli handle ai menù in un array chiamato mymenus. Poi questi menù vengono inseriti nella barra dei menù con le chiamate a *InsertMenu*, il cui primo parametro è lo handle al menù record e il secondo è l'indice del menù prima del quale inserire questo (0 significa inserire per ultimo). Infine si ridisegna la barra dei menù con *DrawMenuBar*. Un trattamento particolare va tenuto con il menù degli accessori di scrivania, in quanto il numero e il tipo di questi può variare a seconda del sistema sotto cui viene eseguita l'applicazione; perciò si preleva dalle risorse il solo titolo del menù e quindi si aggiungono come items i nomi delle risorse di tipo 'DRVR' (driver) che vengono trovati nel file di sistema (chiamata ad *AddResMenu*).

Di seguito vediamo la definizione dei menù nel file di risorse. Il menù di indice 1 è il menù mela, quello sotto cui saranno elencati gli accessori di scrivania, di cui viene solo specificato il titolo con \14 che sta ad indicare il carattere . Nel seguito si possono osservare gli equivalenti di tastiera (i caratteri preceduti dalla barra) e gli item inizialmente disabilitati (quelli preceduti da una parentesi aperta).

```
Type MENU
  ,1
  \14

  ,256
Archivio
    Nuovo/N
    Apri/O
    Chiudi
    Salva/S
    Salva come
    Formato di stampa
    Stampa
    Esci/E

  ,257
Edit
    (Undo/Z
    (-
    Cut/X
    Copy/C
    Paste/V
    (-
    Nuovo record /R
    Modifica record /M
    Cancella - Riattiva /D

  ,258
Ordinamento
    Per nome
    Per cognome
    Per città
    Per Via
    Per CAP
    Per Tel
    Per data

  ,259
Ricerca
    Cerca
```

L'utente, come abbiamo già detto, può scegliere una opzione di un menù in due modi: con il mouse o con l'equivalente di tastiera. Nel primo caso il programma riceverà un evento MouseDown avvenuto nella barra dei menù; nel secondo caso riceverà un evento KeyDown nel cui campo modifiers sarà indicata la pressione del tasto "command". Questi due eventi devono essere considerati equivalenti e quindi si deve rispondere ad essi nello

stesso modo.

Nel nostra programma le azioni corrispondenti alle scelte di menù vengono affidate alla procedura **docommand** cui si passa come parametro un LONGINT nella cui parte alta c'è l'indice del menù scelto e nella parte bassa il numero progressivo dell'item in quel menù. Questo LONGINT viene restituito, nel caso di scelta effettuata con il mouse, dalla funzione *MenuSelect* e , nel caso di equivalenti di tastiera, dalla funzione *MenuKey*. La funzione *MenuSelect* prende il controllo finchè l'utente non rilascia il mouse, esegue tutte le operazioni di visualizzazione dei menù e di evidenziazione degli item selezionati e alla fine restituisce come valore il LONGINT che identifica l'item scelto o 0 se non è stata effettuata nessuna scelta. Il parametro passato a *MenuKey* è il carattere di cui si cerca l'equivalente di tastiera; la funzione restituisce un valore che identifica l'item di menù equivalente.

Vediamo ora le caratteristiche essenziali della procedura **docommand**.

```
PROCEDURE docommand(mResult: LONGINT);

 VAR
  name      : STR255; { Nome dell' accessorio da aprire }
  refnum,    { Numero di riferimento del desk accessory }
  themenu,   { numero del menù selezionato }
  theitem   : INTEGER;{ numero dell'item nel menù }

 BEGIN
  themenu := hiword(mResult);
  theitem := loword(mResult);
  CASE themenu OF

   APPLEMENU:
    BEGIN
     getitem(mymenus[1], theitem, name);
     refnum := opendeskacc(name);
    END;

   FILEMENU:
    CASE theitem OF
     1: donew;             { NUOVO }
     2: doopen;            { APRI }
     3: doclose;           { CHIUDI }
     4: dosave;            { SALVA }
     5: dosaveas;          { SALVA COME }
```

```
  6: dopsetup;          { FORMATO DI SAMPA }
  7: doprint;           { STAMPA }
  8:                    { ESCI }
   BEGIN
    doclose;
    doneflag := TRUE;
   END;
 END;

EDITMENU:
 BEGIN
  { Se non e' una operazione di edit che riguarda un
    desk accessory }
  IF NOT systemedit(theitem - 1)
   THEN
    BEGIN
     SetPort(thewindow);
     ClipRect(prect);
     CASE theitem OF
      1: undo;
      3: mycut;                     { TAGLIA }
      4: mycopy;                    { COPIA }
      5: mypaste;                   { INCOLLA }
     END;                           {* of item case *}
    END;
 END;                               {* of editMenu *}

SORTMENU:
 IF cursort <> theitem
  THEN
   BEGIN
    IF theitem = SORTDATA
{ Non si può fare search se l'ordinamento è per data }
       THEN
      BEGIN
       DisableItem(mymenus[SEARCHID], 0);
      END;

    IF cursort = SORTDATA
     THEN
      BEGIN
       EnableItem(mymenus[SEARCHID], 0);
      END;

    CheckItem(mymenus[SORTID], cursort, False);
    { Aggiorna il tipo di ordinamento corrente }
    cursort := theitem;
    CheckItem(mymenus[SORTID], cursort, TRUE);
    sort;
   END;
```

```
    SEARCHMENU:
      .......
   END;                                {* of menu case *}

   { Ridisegna la menù bar senza menù selezionati }
   hilitemenu(0);
   DrawMenuBar;

  END;                                 {* of DoCommand *}
```

Come si può vedere, inizialmente si estraggono dal parametro l'indice del menù e l'item selezionato, quindi si esegue un'istruzione di CASE su di essi e si eseguono le operazioni associate.

Ancora una volta per quanto riguarda gli accessori di scrivania è necessaria una gestione particolare, dato che non si può sapere a priori quali siano disponibili nel sistema. Nel caso sia stato scelto un item nel menù mela si deve quindi estrarre il nome dell'accessorio con *GetItem* e quindi aprirlo con *OpenDeskAcc.*

Anche nella gestione del menù **Edit** è necessario tener presenti gli accessori di scrivania per permettere il passaggio di dati con le operazioni di *taglia, copia* e *incolla.* Si utilizza a questo scopo la funzione *SystemEdit* che restituisce un valore TRUE se è attualmente attivo un accessorio di scrivania ed esegue su questo le operazioni di edit. Se la finestra attiva non è quella di un accessorio *SystemEdit* ritorna FALSE e quindi si procede a eseguire le operazioni di edit chiamando le nostre procedure.

Nel caso la scelta sia nel menù di ordinamento, si deve semplicemente settare il valore della variabile globale cursort al tipo di ordinamento prescelto e si deve indicare nel menù l'ordinamento corrente con un checkmark. A questo scopo si chiama *CheckItem* prima per togliere (terzo parametro = FALSE) il marker dove era in precedenza e poi per metterlo (terzo parametro = TRUE) sull'item selezionato. Inoltre il nostro programma non permette di fare ricerche se l'ordinamento corrente è per data, in questo caso si deve disabilitare il menù ricerca e viceversa riabilitarlo quando si ritorna ad un altro tipo di ordinamento.

Si utilizzano in questo caso le procedure *DisableItem* e *EnableItem* cui si passa lo handle al record del menù e il numero dell'item da disabilitare; se quest'ultimo parametro è 0 viene disabilitato l'intero menù. Alla fine della procedura docommand è necessario chiamare *HiliteMenu* e *DrawMenuBar* per ridisegnare il titolo del menù selezionato, poichè sia *MenuSelect* che *MenuKey* lo lasciano evidenziato in reverse.

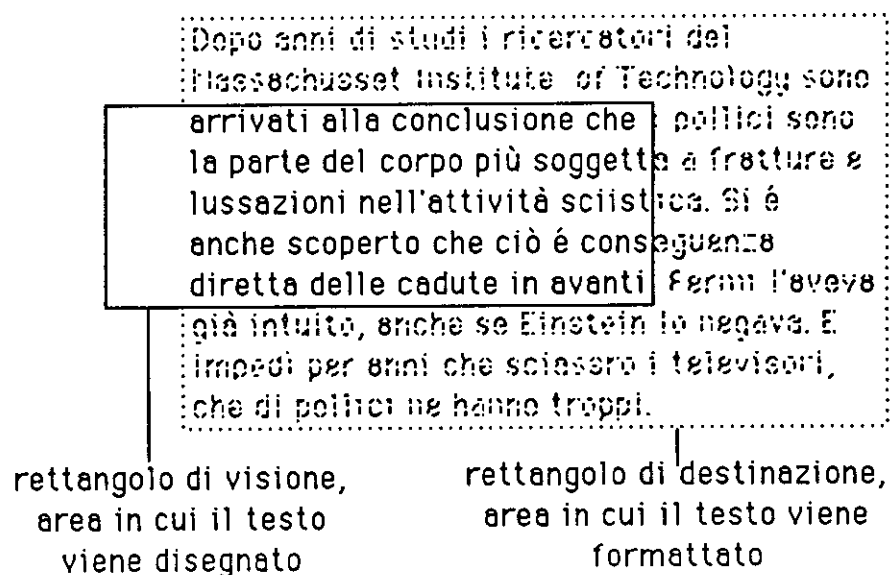
Text Edit

Praticamente ogni applicazione Macintosh, che sia o non sia direttamente interessata al trattamento dei testi, ha occasione di fare delle semplici operazioni di text editing. Le linee guida della User Interface prescrivono certe convenzioni standard per la selezione e le modifiche del testo. In questa sezione ci occuperemo di come si può usare l'insieme di routines di text editing, chiamato TextEdit, che il Toolbox mette a disposizione.

TextEdit fornisce solo le possibilità minime di editing, senza particolari raffinatezze. Lavora con il testo nella forma di sequenza di caratteri, senza struttura interna o informazioni di formattazione. Questo significa che può visualizzare il testo in una singola fonte, stile e dimensione alla volta: non è possibile ad esempio cambiare fonte in mezzo al testo o anche mettere una parola in grassetto.

La struttura su cui TextEdit opera principalmente è lo edit record di tipo TERec. Gli edit records sono oggetti rilocabili che risiedono nello heap cui si fa riferimento tramite uno handle di tipo TEHandle. I campi di un edit record sono molti, ma la maggior parte di questi vengono utilizzati solamente da TextEdit e il programmatore non se ne deve preoccupare. Ogni edit record ha un proprio testo su cui lavora, cui fa riferimento tramite lo handle hText; il testo non è una stringa Pascal ma un array di caratteri di lunghezza variabile, che viene tenuta nel campo TELength dell'edit record ed è un intero senza segno a 16 bit (quindi la lunghezza massima del testo è limitata a 65535 bytes).

A ogni edit record è associata una particolare porta grafica, normalmente una finestra, su cui vengono eseguite le operazioni di editing. Quando si crea un nuovo edit record, a questo viene associata la porta grafica corrente da cui eredita anche le caratteristiche grafiche del testo (txFont, txFace, txMode, txSize). Altri due campi di fondamentale importanza sono il rettangolo di destinazione e quello di visione. Entrambi sono definiti in coordinate locali della porta grafica associata all'edit record, cioè in coordinate relative alla finestra. Il rettangolo di destinazione delimita la zona in cui il testo verrà posto, mentre il rettangolo di visione è quello in cui il testo viene effettivamente disegnato.

Normalmente TextEdit non permetterà al testo di uscire dal rettangolo di destinazione. Quando il testo raggiunge il margine destro del rettangolo di destinazione TextEdit lo manderà automaticamente a capo, senza troncare le parole. E' però possibile indicare a TextEdit di non svolgere questa funzione mettendo il campo crOnly dello edit record a -1. In questo caso il testo cambierà linea solo dopo un carriage return (codice ASCII $0D). La spaziatura verticale del testo è determinata dal campo lineHeight che indica la distanza in pixels tra una linea e l'altra.

I rettangoli di visione e di destinazione

Dato il rettangolo di destinazione e le caratteristiche del testo (fonte, dimensione ecc.), TextEdit può calcolare quante linee occupa il testo e dove sono i cambi di linea. TextEdit tiene questa informazione nel campo lineStarts,che è un array di interi che indica a quale offset rispetto all'inizio del testo avvengono i vari cambi di linea (ad es. lineStarts[5] sarà il numero di caratteri dopo cui finisce la quinta linea). Il numero di linee del testo viene tenuto nel campo nLines dell'edit record.

TextEdit ricalcola automaticamente queste caratteristiche del testo dopo ogni operazione che potrebbe aver modificato i cambi

di linea, come taglia, incolla o inserimento di caratteri da tastiera. I cambi linea possono essere modificati anche da altre operazioni, come il cambiamento della fonte o della larghezza del rettangolo di destinazione. In questi casi il programma può forzare TextEdit a ricalcolare tutto chiamando la routine *TECalText*.

Vediamo ora l'utilizzo delle routines di TextEdit all'interno del programma di esempio. La procedura che ne fa più largo uso è **Fillwindow**, che si occupa di aggiornare il contenuto della finestra degli indirizzi.

```
PROCEDURE fillwindow;

 VAR
  i           : INTEGER;
  textpos, len  : LONGINT;
  acr, asp, astr : STR255;

 BEGIN
  asp := ' ';          { Stringa di un solo blank. }
  acr := ' ';
  acr[1] := CHR(CR);   { Stringa con un solo <return> }

  { Modifica la dimensione del blocco usato per il testo}
  SetHandleSize(hte^^.hText,
               numrec * (SIZEOF(person) + 2));

  { Lock di tutti gli handle coinvolti }
  HLock(Handle(thetabhdl));
  HLock(Handle(theindhdl));
  HLock(Handle(hte));
  HLock(hte^^.hText);

  textpos := 0;

  { Si scrive il contenuto di ogni record accedendovi
    attraverso la scansione sequenziale della tabella di
    indici (TheIndHdl^^) che è ordinata secondo il tipo
    di sort corrente. La variabile "i" assume i valori da
    0 fino a NumRec - 1 che è l'indice della posizione
    della tabella di indici in cui si trova l'ultimo
    indice dei records. }
  FOR i := 0 TO numrec - 1 DO
  { Accesso al record vero e proprio }
   WITH thetabhdl^^[theindhdl^^[i]], hte^^ DO
    BEGIN
     { Si mette un asterisco davanti ai record
```

```
      cancellati o uno spazio davanti a quelli attivi }
    IF deleted
     THEN astr := '*'
     ELSE astr := ' ';

    { poi si scrivono i vari campi, separati da uno
      spazio, in una stringa che termina con Return }
    astr := Concat(astr, asp, nome, asp, cognome, asp,
            citta, asp, via, asp, cap, asp, tel, acr);

    { Si trasferisce direttamente la stringa nell'area
      di Text Edit con BlockMove }
    len := Length(astr);
    BlockMove(POINTER(ORD4(@astr) + 1),
              POINTER(ORD4(hText^) + textpos), len);

    { Aggiorna la posizione di inserimento del testo nel
      record di editing }
    textpos := textpos + len;
   END;

 { Si rilasciano gli Handles precedentemente bloccati }
 HUnLock(Handle(thetabhdl));
 HUnLock(Handle(theindhdl));
 HUnLock(Handle(hte));
 HUnLock(hte^^.hText);

 { Si aggiorna la dimensione effettiva del blocco usato
   da Text Edit per il testo. }
 SetHandleSize(hte^^.hText, textpos);
 hte^^.TELength := textpos;

 { Si ricalcola il testo nella finestra e si fa in modo,
   con InvalRect, che venga ridisegnato tutto il
   contenuto (pRect) }
 TECalText(hte);
 InvalRect(prect);

 TEUpdate(hte^^.ViewRect, hte);
END;
```

Al campo crOnly dell'edit record viene assegnato -1 in fase di inizializzazione, quindi siamo garantiti che gli indirizzi verranno riscritti (da *TEUpdate*) uno per linea, "sbordando" quanto è necessario dal rettangolo di visione (ViewRect), ed andando a nuova linea solo a fine indirizzo, che noi abbiamo segnalato aggiungendo aCr, cioè un "Return" alla fine della stringa.

La procedura che segue serve per selezionare la linea su cui l'utente ha premuto il mouse. Il parametro line contiene l'indice del record selezionato. Utilizzando l'array lineStarts, che contiene le posizioni di inizio delle linee del testo, e *TESetSelect*, che dati due indici, seleziona (cioè riscrive in bianco su nero) la parte di testo compresa, indichiamo sulla finestra il record corrente.

```
PROCEDURE seltheline(line: INTEGER);

  VAR
   da, a     : LONGINT;

  BEGIN
   WITH hte^^ DO
    BEGIN
     da := lineStarts[line];
              {posizione iniziale  della linea "line"}
     a := lineStarts[line + 1];
              {posizione iniziale della linea successiva}

  TESetSelect(da, a - 1, hte);
              {seleziona il testo compreso tra le due
              posizioni}
    END;
  END;
```

La procedura **Scrollbits** è invece quella che viene utilizzata per la gestione degli scroll nella finestra degli indirizzi. Ad essa si fa ricorso ogni volta che l'utente decide di muovere il testo per mezzo dei controlli. Dopo aver calcolato la nuova origine del rettangolo di visione del testo, si calcolano gli spostamenti dh e dv e si ricorre alla routine di TextEdit *TEScroll*, che fa scorrere il testo e lo riaggiorna sulla finestra.

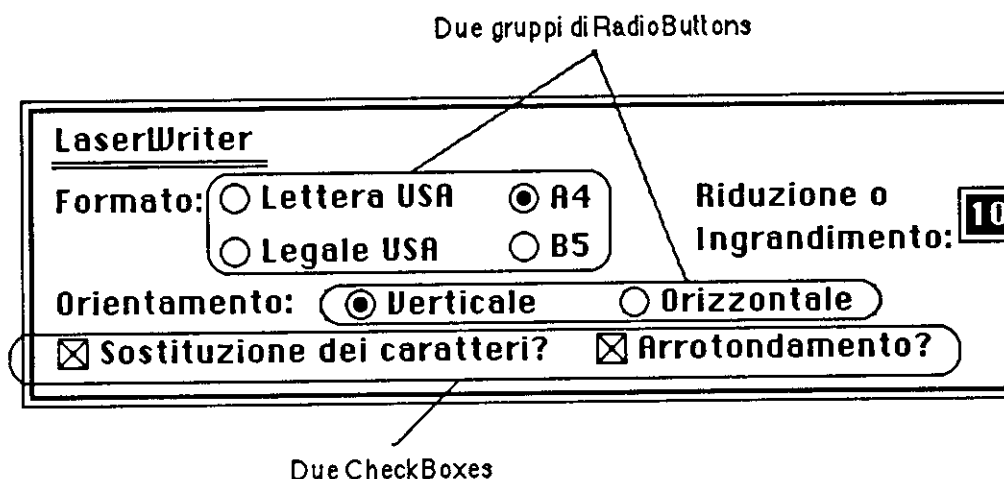
```
PROCEDURE scrollbits;

  VAR
   oldorigin: point;
   dh, dv    : INTEGER;

  BEGIN
   { Salvo la vecchia origine }
   oldorigin := theorigin;
```

```
 { Calcolo la futura origine }
 theorigin.h := hte^^.lineHeight * GetCtlValue(hscroll);
 theorigin.v := hte^^.lineHeight * GetCtlValue(vscroll);

 { Calcolo quanto spostare il contenuto della finestra}
 dh := oldorigin.h - theorigin.h;
 dv := oldorigin.v - theorigin.v;

 { Lo sposto }
 TEScroll(dh, dv, hte);
END;
```

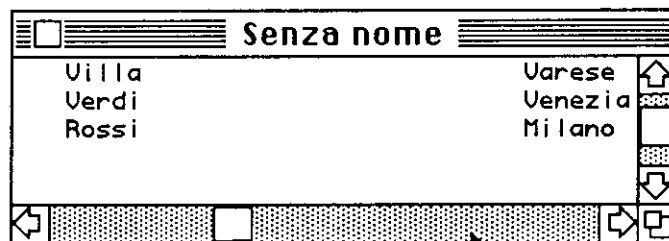
I Controls

Seguendo le linee dell'interfaccia Macintosh, i controlli permettono l'utilizzo del mouse per la scelta di opzioni o per la modifica dei comportamenti dei programmi e degli oggetti che vengono manipolati. La creazione dei controlli avviene per mezzo di routines del Toolbox, mentre il loro comportamento dipende dalla funzione di definizione, che può essere predefinita (per alcuni tipi di controlli) oppure completamente a carico del programmatore che desidera un tipo particolare di comportamento dal proprio controllo. *Inside Macintosh* chiarisce le modalità di creazione e manipolazione dei controlli user-defined, mentre qui ci soffermeremo solo sui controlli predefiniti, categoria cui appartengono tutti quelli utilizzati nel programma.

Due sono i tipi in cui si possono dividere tutti i controlli: i bottoni e i dials. In generale si può dire che il primo tipo comprende i controlli "digitali" (a due stati, on/off), mentre nel secondo sono raggruppati quelli "analogici", che possono assumere più di due valori (pur sempre discreti).

I bottoni predefiniti sono a loro volta suddivisi in tre gruppi: i pushbuttons (che hanno in genere effetto immediato, e talvolta tale effetto dura finchè il bottone viene tenuto premuto) , i checkbox (che servono per scegliere opzioni indipendenti dalle altre) e i radiobuttons (che prendono il loro nome dai bottoni di selezione dei canali predefiniti nelle autoradio: non più di uno per gruppo può essere in posizione "on"; vengono utilizzati per la scelta di una tra due o più opzioni alternative). Il comportamento di tutti questi bottoni è identico, mentre cambia, ovviamente, la forma in cui si presentano sullo schermo e in cui vengono segnalati gli stati di on/off.

RadioButtons e CheckBoxes

PushButtons

Del secondo tipo di controlli, i dials, vi è un solo sottotipo predefinito, le barre di scroll, di cui però viene fatto uso sia nel nostro programma, sia in tutti i programmi in cui si vogliano utilizzare delle finestre come "ingranditori" di parti di documenti su cui scorrano a piacimento. Diversamente dai bottoni, le barre di scroll sono suddivise in più zone dal comportamento differente: le due frecce (in alto e in basso), che fanno spostare il contenuto della finestra di una riga, le regioni di page-up e page-down, che spostano di una pagina intera, e la scroll-box o thumb, che serve da controllo "analogico" della posizione della finestra rispetto al documento complessivo (dice cioè "a che punto siamo" del testo).

Una finestra con due barre di controllo

La creazione di un controllo avviene per mezzo della funzione *GetNewControl* (ctlID,Window), che legge dal file di risorse le caratteristiche del controllo ctlID, lo alloca, lo lega alla finestra Window e ritorna l'handle al nuovo controllo. Nelle risorse occorrerà definire il rettangolo in cui dovrà apparire (relativo all'origine della finestra che lo possiede), il titolo (che nel caso dei bottoni è il testo che viene scritto dentro o a fianco del controllo), i valori iniziale, minimo e massimo (utili nel caso di controllo "analogico"), se sia visibile o no e infine l'ID della procedura di definizione (che nel caso dei controlli predefiniti è 0 per i PushButtons, 1 per i checkbox, 2 per i RadioButton e 16 per le barre di scroll). Il testo in tutti i controlli viene scritto in fonte Chicago, a meno che non si richieda esplicitamente l'uso della fonte associata alla finestra su cui devono apparire; ciò si ottiene aggiungendo la costante UseWFont (8) all'ID di definizione.
Le risorse di tipo control vengono definite come di seguito:

```
Type CNTL
  ,256                          ;; ID di vscroll
  vertical scroll bar           ;; titolo
  -1 395 236 411                ;; rettangolo di
                                ;; definizione
  Visible                       ;; subito visibile
  16                            ;; barra di controllo
  0                             ;; refCon
  0 50 0                        ;; min max iniziale

  ,257
  horizontal scroll bar
  235 -1 251 396
  Visible
  16
  0
  0 50 0
```

L'alternativa all'uso delle risorse è data da *NewControl*, a cui vanno passate le stesse informazioni viste nel caso dell'uso delle risorse.

La rimozione dei controlli viene poi eseguita dalla routine *DisposeControl*, oppure implicitamente nel momento in cui la finestra cui appartengono viene chiusa o rimossa.

L'aspetto di un controllo viene gestito atteverso la procedura *HiliteControl*, cui viene passato il codice della parte del controllo da evidenziare. In particolare 0 riporta allo stato normale il controllo, mentre 255 lo disattiva.

Un pushButton inattivo e uno attivo

Nella procedura che segue si fa uso di due routines di Toolbox riguardanti i controlli, *SetCtlMax*, che assegna il valore massimo che può raggiungere un controllo, e *DrawControls*, che ridisegna tutti i controlli legati ad una finestra.

```
PROCEDURE adjustscrollbars;

 VAR
  maxctlval: INTEGER;
  trect    : Rect;

 BEGIN
  WITH hte^^ DO
   BEGIN
    maxctlval := nLines - 1;
    {Il max. del control è il numero di linee di testo}
    IF maxctlval < 0
     THEN maxctlval := 0;
    {Assegna allo scroll verticale il nuovo massimo}
    SetCtlMax(vscroll, maxctlval);
    {E ridisegna i controlli}
    DrawControls(thewindow);
    trect := thewindow^.portRect;
```

```
      trect.left := trect.right - 16;
      {Segnala che va ridisegnato lo scroll}
      InvalRect(trect);
      { Verrà ridisegnata anche la scroll-bar orizzontale,
      ma il valore resta invariato }
      trect := thewindow^.portRect;
      trect.top := trect.bottom - 16;
      InvalRect(trect);
    END;
  END;
```

Il valore di un controllo viene ottenuto tramite la funzione *GetCtlValue*, mentre può essere definito per mezzo di *SetCtlValue*. Queste due routines sono di particolare importanza nell'utilizzo delle barre di scroll, perchè permettono di aggiornare la situazione dopo l'intervento dell'utente.

Le due routines **Scrollup** e **Scrolldown** utilizzano *SetCtlValue* per aggiornare il valore della barra di scroll (e riposizionare automaticamente il thumb). Poichè il valore minimo è 0 e il massimo è pari al numero di linee nel testo, ogni passo di scroll è pari ad una linea.

```
PROCEDURE scrollup(whichcontrol: controlhandle;
                    theCode: INTEGER);

  BEGIN
   IF theCode = inUpButton
    THEN
     BEGIN
      SetCtlValue(whichcontrol,
                  GetCtlValue(whichcontrol) - 1);
      scrollbits
     END
  END;

 PROCEDURE scrolldown(whichcontrol: controlhandle;
                      theCode: INTEGER);

  BEGIN
   IF theCode = inDownButton
    THEN
     BEGIN
      SetCtlValue(whichcontrol,
                  GetCtlValue(whichcontrol) + 1);
      scrollbits
     END
```

```
  END;
```

La procedura **pagescroll**, che segue, serve per rispondere ai click del mouse nelle due zone della barra di scroll, denominate pageup e pagedown. Un click in tale zona equivale, secondo lo standard dell'interfaccia Macintosh, allo scroll di una pagina del contenuto della finestra. Questa procedura viene richiamata ad ogni evento di mousedown in una delle due zone, e rimane attiva finchè l'utente non rilascia il pulsante del mouse.
Durante questo periodo, se il mouse rimane premuto sulla stessa zona (per verificare ciò si usa *TestControl*), il valore della barra viene modificato di una certa quantità, definita al livello superiore.

```
PROCEDURE pagescroll(code, amount: INTEGER);
  VAR
   mypt      : point;

  BEGIN
   REPEAT
    GetMouse(mypt);
    IF TestControl(whichcontrol, mypt) = code
     THEN
      BEGIN
       SetCtlValue(whichcontrol,
                   GetCtlValue(whichcontrol)  + amount);
       scrollbits
      END
   UNTIL NOT StillDown;
  END;
```

Un ultimo problema associato ai controlli nasce nel momento in cui la finestra cambia dimensioni. Allora diventa necessario ridisegnarli con le dimensioni corrette.

A tale scopo definiamo la procedura **movescrollbars**, in cui compaiono quattro nuove procedure. *Hidecontrol* e *Showcontrol* hanno il compito di modificare la visibilità del controllo (ma se il controllo è "sotto" qualcos'altro o la finestra è invisibile, *Showcontrol* non può farci nulla); *Movecontrol* sposta l'angolo in alto a sinistra del controllo nella nuova posizione (sempre comunque locale alla finestra cui appartiene); *Sizecontrol* infine aggiorna le dimensioni del controllo.

```
PROCEDURE movescrollbars;

 BEGIN
  WITH thewindow^.portRect DO
   BEGIN
    {Nascondi per il momento il vecchio controllo}
    HideControl(vscroll);
    {Spostalo nella nuova posizione}
    MoveControl(vscroll, right - 15, top - 1);
    {Aggiorna le sue dimensioni}
    SizeControl(vscroll, 16, bottom - top - 13);
    {Dopo averlo aggiornato rendilo di nuovo visibile}
    ShowControl(vscroll);

    {Idem per l'altro scroll...}
    HideControl(hscroll);
    MoveControl(hscroll, left - 1, bottom - 15);
    SizeControl(hscroll, right - left - 13, 16);
    ShowControl(hscroll)
   END
 END;
```

Ulteriori commenti sui controlli, in special modo sui bottoni, verranno fatti nella sezione riguardante le finestre di dialogo, dove la loro funzione risulta insostituibile per garantire l'interattività tra il programma e l'utente.

I Dialog

L'esecuzione di un comando dell'utente spesso necessita di ulteriori informazioni, al di là di quelle implicite nel richiamo di un certo item di menù in un certo ambiente (un esempio è dato dal comando di **Salva come...** nel menù **File**, per il quale occorre sapere anche il nome del nuovo file che si va a creare). Oppure, possono verificarsi situazioni in cui il programma deve avvertire l'utente di particolari condizioni di errore o di pericolo per i suoi dati.

Allo scopo di favorire l'interazione tra programma ed utente sono stati introdotti i Dialog. Si tratta di finestre particolari, suddivise in tre categorie, a seconda dell'uso che ne viene fatto e del comportamento di fronte alle azioni dell'utente.

Il primo tipo di dialog è costituito dagli Alert, che vengono utilizzati per emettere messaggi di errore o altre informazioni sullo stato del programma. Quando un alert è visibile, è sempre sopra tutte le altre finestre e non permette all'utente altro che un click in un pushbutton (o in un'altra parte) per rilasciarla.

Ad un livello di interattività superiore agli alert stanno i Modal Dialog, cosiddetti perchè costringono l'utente in un "modo", cioè gli permettono solo un certo tipo di azioni; più specificamente, gli permettono di agire attraverso mouse e tastiera solo sulla finestra di dialogo. Fino al momento in cui l'utente non preme il mouse in uno dei bottoni predisposti per l'uscita, ogni click fuori dalla finestra viene segnalato con un beep e non ha alcun effetto.

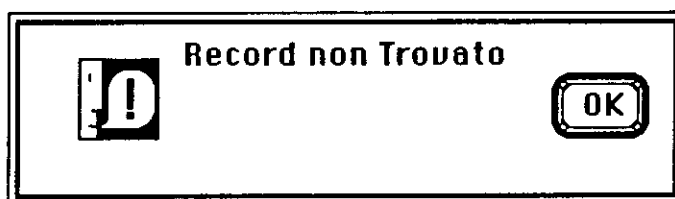
Infine vi è il terzo tipo, i Modeless Dialog, quelli cioè che possono essere attivati, deattivati e mossi sullo schermo senza problemi. In questo caso essi possono essere rilasciati per mezzo della close box, esattamente come le ordinarie finestre di documenti.

La creazione di un Modeless o Modal dialog avviene per mezzo delle routines *GetNewDialog* (se si utilizzano le risorse di tipo 'DLOG') o *NewDialog* (se i parametri vengono forniti a livello di programma). La rimozione viene invece effettuata dalle routines *CloseDialog* o *DisposDialog*, a seconda che il record della finestra venga fornito da programma o richiesto al Toolbox. Gli Alert

vengono invece creati direttamente dal Toolbox utilizzando le risorse di tipo 'ALRT'.

Una risorsa di tipo dialog è suddivisa in due parti: una dichiarazione riguardante la finestra vera e propria (di tipo 'DLOG' o 'ALRT') ed una riguardante i vari elementi (items) che dovranno comparire sulla finestra, di tipo 'DITL' (Dialog ITem List).

Questi ultimi possono essere di diversi tipi:

- testo statico, indicato nelle risorse come StatText. E' la forma tipica in cui vengono passati i messaggi all'utente. La stringa può contenere fino a quattro segnalatori, ^0, ^1, ^2, ^3, al cui posto possono essere inserite stringhe arbitrarie per mezzo della procedura *ParamText*, cui vanno passate le stringhe che sostituiranno i segnalatori.
- icone e figure (IconItem e PicItem). Le icone di maggiore importanza sono quelle utilizzate nei vari sottotipi di alert:
  - note: serve per sottoporre all'attenzione dell'utente qualche informazione particolare
  - caution: avverte di qualche anomalia più grave o chiede informazioni ulteriori per continuare
  - stop: viene usato in caso di guai seri o problemi che impediscono la normale continuazione del programma.
- controlli, sia standard che definiti dall'utente
- rettangoli di editing
- altri elementi definiti dall'utente con sue procedure.

Un alert box

Degli elementi che compaiono nella Item List è importante ricordare il numero d'ordine (a partire da 1), perchè è ad esso che si farà sempre riferimento nelle operazioni di lettura/scrittura del

dialog. Inoltre particolare interesse ha il primo elemento in lista (i primi due nel caso di un Alert): infatti la pressione di un tasto di conferma (Return o Enter) in presenza di un dialog equivale al click in quell'elemento (detto di default). Occorre allora che ogni dialog contenga almeno un bottone per il suo rilascio, e che il primo elemento della lista degli elementi sia il bottone più "sicuro", quello cioè che si consiglia all'utente di usare.

Nel nostro programma il dialog più importante è quello utilizzato per la creazione e la modifica dei record di indirizzi.
La procedura **openrecord** si fa carico delle opportune chiamate per la creazione del dialog e l'aggiornamento del record:

```
{ Apre il dialog per modificare il record. Ind è l'indice
del record nella PersTab. }
 PROCEDURE openrecord(ind: INTEGER);

  BEGIN
  {non aprire dialog se non ci sono record da aggiornare}
   IF numrec > 0
    THEN
     BEGIN
      newdlog;   {crea il dialog}
      filldlog(ind); {riempi i campi con il record
                      indicato}
      dlognew := False; {non è un dialog per un nuovo
                          record}
      dlogloop; {esegui le operazioni necessarie finchè
                 non viene chiuso il dialog}
     END;
  END;
```

La procedura che segue carica dalle risorse, con *GetNewDialog*, il dialog 256, che è appunto il dialog per la creazione/modifica dei record.

Il parametro NIL indica al Toolbox di ricavarsi da solo lo spazio di memoria in cui allocare la finestra, mentre il POINTER(-1) indica a QuickDraw di disegnare la finestra sopra a tutte le altre.
La variabile thedlog è definita a livello globale ed è di tipo DialogPtr, che è equivalente ad un WindowPtr.

```
PROCEDURE newdlog;

  BEGIN
   thedlog := GetNewDialog(256, NIL, POINTER( - 1));
  END;
```

Qui di seguito vengono illustrate le risorse relative al dialog di modifica e alla sua Item List.

```
TYPE DLOG
  ,256                            ;; Id del dialog
  Dialog di modifica              ;; titolo
  40 40 280 440                   ;; rettangolo di definizione
  Visible NoGoAway                ;; visibile, senza close box
  1                               ;; tipo:dialog box standard
  0                               ;; refCon
  256                             ;; ID della item list


TYPE DITL
,256      ;; Item list 256: elem. della dialog di modifica
  20      ;; vi sono 20 elementi

BtnItem Enabled   ;; un bottone inizialmente abilitato
10 340 30 390     ;; in questo rettangolo
OK                ;; testo del bottone

BtnItem Enabled   ;; un bottone
40 340 60 390
Esci              ;; per concludere il dialogo

BtnItem Enabled   ;; un altro
190 10 210 90
Prossimo          ;; per passare al record successivo

BtnItem Enabled   ;; e uno
190 110 210 190
Precedente        ;; per tornare indietro

BtnItem Enabled   ;; uno per svuotare il dialog
190 210 210 290
Nuovo             ;; e scriverne un nuovo record

BtnItem Enabled   ;; e un bottone
190 310 210 390
Cancella         ;; per cancellare(o riattivare) il record

StatText         ;; del testo statico
```

```
10 10 30 85        ;; in questo rettangolo
Nome

StatText
40 10 60 85
Cognome

;; e così via per tutti gli altri campi del record...

*Nome              (questo è un commento)
EditText     ;; un rettangolo per l'editing del campo nome
10 90 30 330

*Cognome
EditText
40 90 60 330

;; e così via per tutti gli altri campi del record...

BtnItem Enabled     ;; ancora un bottone
70 340 90 390
Scrivi              ;; per aggiornare il record senza uscire
                    ;; dal dialog

StatText                    ;; ancora del testo statico
220 310 240 390
* attivo/cancellato         ;;non ancora specificato
```

Analizziamo ora le altre procedure utilizzate in **openrecord**: dopo aver creato e disegnato la finestra, occorre riempire i vari campi con gli elementi del record da aggiornare. **Filldlog** si occupa appunto di questa incombenza, utilizzando due nuove routines di Toolbox: la prima, *GetDItem*, dato un dialog ed un numero di item (corrispondente alla sua posizione nella item list), ritorna in tre parametri di tipo var il tipo dell'elemento, un handle al record di informazioni relative e il rettangolo in cui compare; la seconda, *SetIText*, dato un handle ad un elemento, ne aggiorna il campo testo. Allora, per poter inserire le stringhe da modificare, ci occorreranno gli handles ai singoli elementi.

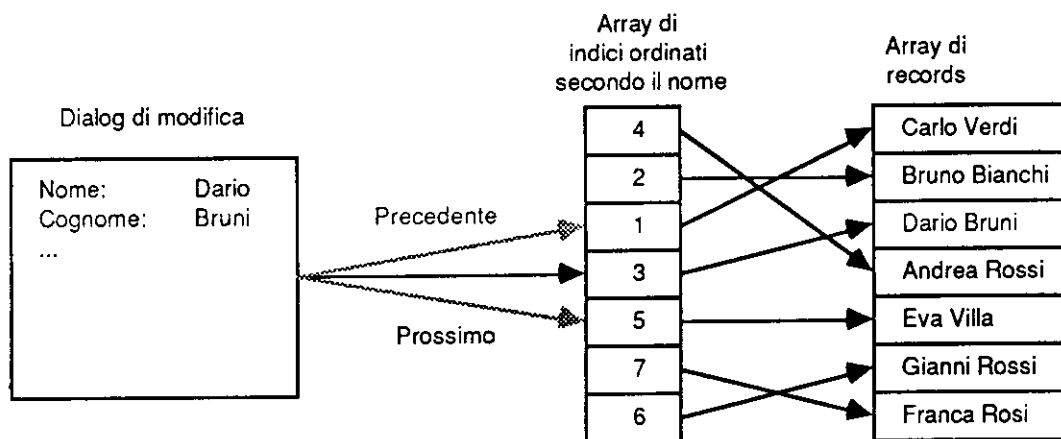
```
PROCEDURE filldlog(ind: INTEGER);

  VAR
   taip      : INTEGER;
   itemhdl   : Handle;
   box       : Rect;
```

```
    astr      : STR255;

  BEGIN
    HLock(Handle(thetabhdl));

    WITH thetabhdl^^[ind] DO
     BEGIN
      {leggi le informazioni riguardanti l'item 13 (nome)}
      GetDItem(thedlog, 13, taip, itemhdl, box);
      {e aggiorna il testo con la stringa nome del record}
      SetIText(itemhdl, nome);
      {e così via per gli altri campi del record...}
      .........
      .........

      { Quindi aggiorna lo stato di Cancellato/Attivo
        usando la AdjFlagIfDel (vedi sotto)}

      {deleted è il flag del record
      (True=cancellato, False=attivo)}
      adjflagifdel(deleted);

     END;

    { Infine aggiorna la situazione dei due bottoni di
      scorrimento sul file, prox e prec, richiamando la
      procedura shiftrec, con spostamento zero   }
    shiftrec(0);

    HUnLock(Handle(thetabhdl));
  END;
```

La procedura **Adjflagifdel**, in base al flag "Deleted" modifica il testo del bottone di cancellazione e la scritta Attivo / Cancellato nel dialog.

```
 PROCEDURE adjflagifdel(deleted: Boolean);

  VAR
   taip      : INTEGER;
   itemhdl   : Handle;
   box       : Rect;

  BEGIN

   { Preleva l'handle al testo dell'item 20,
     Attivo/Cancellato }

   GetDItem(thedlog, 20, taip, itemhdl, box);
```

```
IF deleted
 THEN SetIText(itemhdl, 'Cancellato')
 ELSE SetIText(itemhdl, 'Attivo');

{ Preleva l'handle all'elemento 6, il bottone
  Cancella/Riattiva}
GetDItem(thedlog, 6, taip, itemhdl, box);
{Trattandosi di un control, bisogna cambiare il tipo
 dell'handle usato da GetDItem in un controlhandle, e
 utilizzare la procedura SetCTitle, per cambiare il
 titolo del controllo}

IF deleted
  THEN SetCTitle(controlhandle(itemhdl), 'Riattiva')
  ELSE SetCTitle(controlhandle(itemhdl), 'Cancella');
END;
```

Per muovere il dialog sul file degli indirizzi, (come un proiettore di diapositive sposta il caricatore, per "accedere" alle varie immagini),

Come si sposta il dialog sul file degli indirizzi

usiamo la procedura **shiftrec**, che, inoltre, si accorge quando stiamo modificando il primo dei record (e quindi non possiamo più arretrare) o l'ultimo (e non si può più avanzare) e disabilita e riabilita opportunamente i due bottoni prossimo e precedente.

```
PROCEDURE shiftrec(offset: INTEGER);

 VAR
  taip      : INTEGER;
  itemhdl   : controlhandle; {Handle all'item del dialog}
  box       : Rect;
 BEGIN
  { Incrementa l'indice del record corrente }
  currecind := currecind + offset;

  IF currecind >= numrec - 1          { Ultimo record }
   THEN
    BEGIN
     { Disabilita il button "prossimo" del dialog }
     currecind := numrec - 1;
     GetDItem(thedlog, 3, taip, Handle(itemhdl), box);
     HiliteControl(itemhdl, 255);
    END;

  IF currecind <= 0                   { Primo record }
   THEN
    BEGIN
     currecind := 0;
     { Disabilita il button "precedente" del dialog }
     GetDItem(thedlog, 4, taip, Handle(itemhdl), box);
     HiliteControl(itemhdl, 255);
    END;

  IF currecind > 0                    { Non è il primo }
   THEN
    BEGIN
     { Abilita il button "precedente" del dialog }
     GetDItem(thedlog, 4, taip, Handle(itemhdl), box);
     HiliteControl(itemhdl, 0);
    END;

  IF currecind < numrec - 1           { Non è l'ultimo }
   THEN
    BEGIN
     { Abilita il button "prossimo" del dialog }
     GetDItem(thedlog, 3, taip, Handle(itemhdl), box);
     HiliteControl(itemhdl, 0);
    END;
 END;
```

Da notare in **shiftrec** il cambiamento di tipo (o casting) inverso (ma assolutamente equivalente: è solo una questione di comodità) rispetto a quello utilizzato in precedenza.

La terza procedura richiamata da **openrecord** è **dlogloop**.

Come dice anche il nome, essa continua a trattare gli eventi relativi al dialog, finchè l'utente non rilascia volontariamente la finestra (premendo uno dei due bottoni "OK" ed "esci"). A tale scopo utilizza la procedura di Toolbox *ModalDialog*, che fa sì che la finestra attiva assuma un comportamento "modale" e termina ad ogni evento significativo ritornando il numero dell'item del dialog che è stato selezionato dall'utente. *ModalDialog* si prende carico di tutti gli eventi e li tratta in modo standard, in particolare si cura di attivazione, disattivazione e aggiornamento della finestra, degli input nei text box, dei click del mouse nei controlli e dei click esterni alla finestra (segnalandoli con un beep). E' possibile definire a livello di programma dei comportamenti diversi, scrivendo un procedura di filtro degli eventi e passandola come parametro a *ModalDialog.*

```
PROCEDURE dlogloop;

 VAR
  theitem   : INTEGER;

 BEGIN
  REPEAT
   ModalDialog(NIL, theitem); {passando NIL utilizziamo
                               il comportamento standard}
   {Tratta l'evento relativo all'item selezionato}
   dlogdptch(theitem);
  UNTIL theitem <= 2; {finchè non viene schiacciato uno
            dei due bottoni di uscita (1=OK o 2=esci)}

  DisposDialog(thedlog);  {Rimuovi la finestra e rilascia
                           la memoria utilizzata}
  { Infine riordina i record e visualizzali nuovamente }
  sort;    {vedi sezione relativa}
 END;
```

In **dlogdptch** vengono eseguite le operazioni relative all'attivazione di un item nel dialog. Vengono ignorati nell'istruzione di case gli item di testo statico e dinamico, in quanto, nel secondo caso, già *ModalDialog* provvede all'aggiornamento del testo.

```
PROCEDURE dlogdptch(item: INTEGER);

  VAR
   taip      : INTEGER;
   itemhdl   : controlhandle;
   box       : Rect;

  BEGIN
   CASE item OF
    1:                               { OK }
     BEGIN
      IF dlognew               {Siamo in inserimento?}
       THEN insert             {inseriscilo}
       ELSE writerec           {sostituisci il vecchio}
      dlognew := False;        {non siamo più in inserimento}
     END;
{2: non far nulla, serve solo per uscire dal loop}
    3:
     BEGIN                           { PROSSIMO }
      dlognew := False;
      shiftrec(1); {Passa al successivo aggiornando i
                    bottoni prox/prec}
      filldlog(theindhdl^^[currecind]); {Carica il record
                                         nel dialog}
     END;
    4:
     BEGIN                           { PRECEDENTE }
      dlognew := False;
      shiftrec( - 1);
      filldlog(theindhdl^^[currecind]);
     END;

    5: newrec;                       { NUOVO }
       {vedi sotto}
    6:                               {CANCELLA/RIATTIVA}
     BEGIN
 { Inverte il valore del flag nel record corrente ed
   aggiorna l'indicatore Attivo/Cancellato }
      WITH thetabhdl^^[theindhdl^^[currecind]] DO
       BEGIN
        deleted := NOT deleted;
        adjflagifdel(deleted);
       END;
      dirty := TRUE; { Il documento è stato modificato! }
     END;

    19:                              { SCRIVI }

     { Se il record è nuovo lo si aggiunge (NewRec)
       altrimenti si aggiorna il record nella tabella, si
       passa al successivo e lo si visualizza }
```

```
      IF dlognew
       THEN newrec
       ELSE
        BEGIN
         writerec;
         shiftrec(1);
         filldlog(theindhdl^^[currecind]);
        END;
    END;
   END;

 PROCEDURE writerec; { Aggiorna il record corrente }

  BEGIN
   fillrec(theindhdl^^[currecind]);
  END;
```

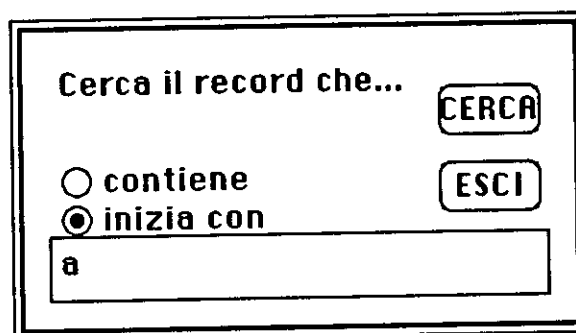
In **fillrec**, che segue, viene utilizzata un'altra routine di Toolbox, *GetIText*, che dato un handle ad un edit box, ritorna la stringa in esso contenuta. Tale stringa viene "normalizzata", aggiungendole degli spazi per riportarla alla lunghezza di definizione (29 per nome, cognome, 9 per il CAP, ecc.) e infine trasferita nel record opportuno.

```
 PROCEDURE fillrec(ind: INTEGER);

  VAR
   taip      : INTEGER;
   item      : Handle;
   box       : Rect;
   astr      : STR255;

  BEGIN
   HLock(Handle(thetabhdl));

   WITH thetabhdl^^[ind] DO { Usa il record appropriato }
    BEGIN

    { Per ogni text box: mette il testo in aStr
      prelevandolo dal Dialog-Item, lo completa con spazi
      usando BlankFill e lo assegna al campo del record }

     {Prendi l'handle all'item}
     GetDItem(thedlog, 13, taip, item, box);
     {Prendi la stringa di testo}
     GetIText(item, astr);
     {Aggiungi gli spazi}
     blankfill(@astr, SIZEOF(nome));
```

```
     {e assegnala al campo opportuno}
     nome := astr;
     .....

     { Nel campo data scrive il valore dell'orologio
       interno (i secondi trascorsi dal 1/1/1904) }
     GetDateTime(data);

     dirty := TRUE; { Il documento è stato modificato! }
    END;
   HUnLock(Handle(thetabhdl));
  END;
```

Nel caso in cui l'utente prema il bottone **Nuovo**, occorre salvare le informazioni eventualmente introdotte prima, e poi "sbiancare" il dialog, ricordando che il record andrà inserito come nuovo e non sovrascritto.

```
PROCEDURE newrec;

 VAR
  i         : INTEGER;
  taip      : INTEGER;
  item      : Handle;
  box       : Rect;

 BEGIN
  IF dlognew  { Si è gia' in inserimento }
   THEN
    BEGIN
     insert;
    END
   ELSE
    {se non lo si era, si passa in modo "inserimento"}
    dlognew := TRUE;

  { Svuota i campi editabili del dialog }
  FOR i := 13 TO 18 DO
   BEGIN
    GetDItem(thedlog, i, taip, item, box);
    {Assegna al testo dell'item la stringa nulla}
    SetIText(item, '');
   END;
  shiftrec(0);              {e aggiorna i bottoni}
 END;

PROCEDURE insert; { Aggiunge un nuovo record in tabella}
```

```
  BEGIN
  {Se è il primo di un nuovo documento, riabilita i menù}
   IF numrec = 0
    THEN
     BEGIN
      EnableItem(mymenus[EDITID], 3);
      EnableItem(mymenus[EDITID], 4);
      EnableItem(mymenus[EDITID], 8);
      EnableItem(mymenus[EDITID], 9);
      EnableItem(mymenus[SORTID], 0);
      EnableItem(mymenus[SEARCHID], 0);
      DrawmenuBar;
     END;

   { Incrementa NumRec, usalo per aggiornare le dimensioni
     delle tabelle e aggiorna la tabella degli indici }
   numrec := numrec + 1;

SetHandleSize(Handle(thetabhdl), numrec * SIZEOF(person));
SetHandleSize(Handle(theindhdl),numrec * SIZEOF(INTEGER));

 {aggiungilo nella tabella degli indici}
theindhdl^^[numrec - 1] := numrec - 1;

 { Scrivi i dati nel record }
   thetabhdl^^[numrec - 1].deleted := False;
   fillrec(numrec - 1);

  END;
```

Il dialog di ricerca

Un altro dialog viene utilizzato per il controllo delle richieste di ricerca. Esso compare come risorsa 258 e quella che segue è la sua Item List (258).

```
TYPE DITL
,258
  6                        ;; sei elementi

  BtnItem Enabled          ;; un bottone
  20 160 40 200
CERCA                      ;; per avviare la ricerca

  BtnItem Enabled           ;; e uno per uscire
  50 160 70 200
ESCI

  StatText Disabled      ;; a che cosa serve questo dialog?
  10 10 40 150
Cerca il record che...

  RadioItem Enabled      ;; un radiobutton
  50 10 62 140
contiene               ;; per controllare il tipo di ricerca

  RadioItem Enabled      ;; e un altro per
  65 10 77 140
inizia con             ;; fornire l'alternativa


  EditText               ;; e finalmente il text box in cui
  80 10 100 200          ;; scrivere la stringa da cercare
```

Quella che segue è la parte di **docommand** che si riferisce ai click nel menù **Ricerca**. Nel flag globale issearchinit viene ricordato il tipo di ricerca (stringa all'inizio del campo / stringa contenuta).

```
  SEARCHMENU:
     BEGIN
      {Attiva il dialog di ricerca}
      searchdlog := GetNewDialog(258, NIL, POINTER( - 1));

      { e assegna al text box l'ultima stringa di ricerca
      utilizzata (inizialmente nulla)}
      GetDItem(searchdlog, 6, taip, itemhdl, box);
      SetIText(itemhdl, searchstr);

      {Aggiorna l'aspetto dei due radiobutton}
      { Prima CONTIENE...}
      GetDItem(searchdlog, 4, taip, itemhdl, box);
      IF issearchinit
```

```
        THEN SetctlValue(controlhandle(itemhdl), 0)
        ELSE SetctlValue(controlhandle(itemhdl), 1);

       { E poi INIZIA CON ...}
       GetDItem(searchdlog, 5, taip, itemhdl, box);
       IF issearchinit
        THEN SetctlValue(controlhandle(itemhdl), 1)
        ELSE SetctlValue(controlhandle(itemhdl), 0);

       {Dopo aver sistemato il dialog gestisci le richieste
        dell'utente finchè non schiaccia ESCI (item n. 2)}
       REPEAT
        ModalDialog(NIL, itemhit);
        searchdptch(itemhit);
       UNTIL itemhit = 2;

       {Infine cancella il dialog e rilascia la memoria
        occupata}
       DisposDialog(searchdlog);
      END;
```

Come in **dlogdptch**, anche in **searchdptch** ci si occupa delle attivazioni degli elementi del dialog, trascurando quelle inutili (nel caso dello static text) o quelle già gestite da *ModalDialog* (nel caso del text box).

```
PROCEDURE searchdptch(item: INTEGER);

 VAR
  ind       : INTEGER;
  taip      : INTEGER;
  itemhdl   : controlhandle;
  box       : Rect;

 BEGIN
  CASE item OF
   1:            {CERCA}
    BEGIN
     GetDItem(searchdlog, 6, taip, Handle(itemhdl), box);
     {Carica in searchstr la stringa da cercare}
     GetIText(Handle(itemhdl), searchstr);
     ind := dosearch; {Esegue la ricerca}
     IF ind >= 0    {record trovato}
      THEN
       BEGIN
        {nuovo indice del record corrente}
        currecind := ind;
       {Aggiorna il valore della barra di controllo
```

```
        verticale}
        SetctlValue(vscroll, ind);
        scrollbits; {Risistema la finestra}
        {Seleziona la linea corrispondente al record
         trovato}
        seltheline(ind);
       END
      ELSE              { Record non Trovato }
       BEGIN
        {Emetti un Alert box, per avvisare l'utente}
        ind := StopAlert(259, NIL);
        {Risistema il testo della finestra}
        TEUpdate(prect, hte)
       END;
    END;

   4:      {CONTIENE}
    BEGIN
     {Abilita il bottone CONTIENE}
     GetDItem(searchdlog, 4, taip, Handle(itemhdl), box);
     SetctlValue(itemhdl, 1);
     {E disabilita INIZIA CON}
     GetDItem(searchdlog, 5, taip, Handle(itemhdl), box);
     SetctlValue(itemhdl, 0);
     {Aggiorna il flag di ricerca}
     issearchinit := False;
    END;

   5:     {INIZIA CON... è simmetrico}
  END;                                 { CASE }
 END;
```

In **searchdptch** abbiamo fatto uso anche di un alert di stop, per avvertire del fallimento della ricerca. L'alert viene gestito completamente dalla funzione *StopAlert*, che dato l'ID della risorsa in cui è il descrittore (di tipo ALRT) e un puntatore alla procedura di filtro (NIL se si vuole il comportamento standard), crea l'alert e si fa carico di tutti gli eventi, finchè l'utente non preme il mouse in un elemento abilitato, di cui ritorna il numero d'ordine nella Item List.

La finestra di alert viene definita nelle risorse, specificando, oltre al rettangolo in cui dovrà comparire e all'ID della Item List, le fasi di avvertimento.

E' infatti possibile definire fino a quattro diversi livelli di "richiamo" dell'utente, tre per le prime tre volte e il quarto per tutte quelle

successive. Per ogni fase è possibile definire se l'alert deve apparire sullo schermo, il tipo di beep emesso e quale dei primi due elementi della Item List sia quello di default. Ogni singola fase viene rappresentata con un semi-byte (quattro fasi, due bytes), in cui:

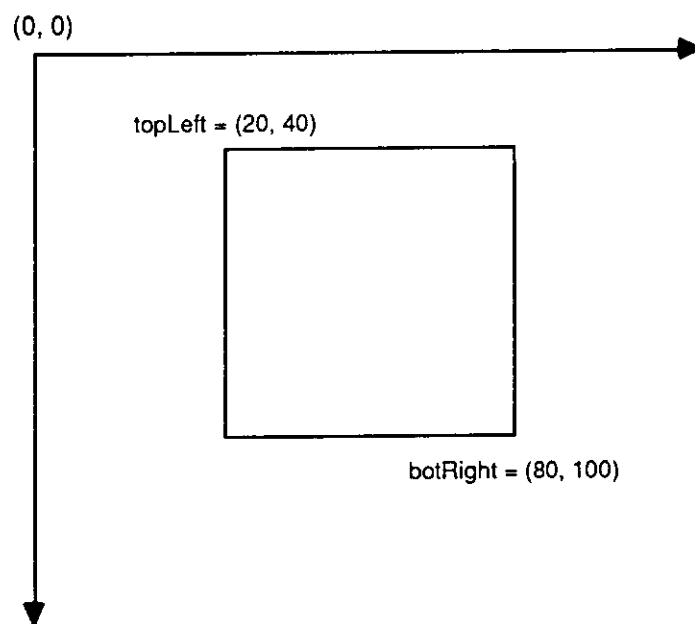
- i due bit meno significativi indicano il tipo di beep (da 0 a 3)
- il terzo se l'alert vada disegnato (0 no, 1 sì)
- il quarto il bottone di default (0 per il primo, 1 per il secondo)

```
Type ALRT
,257                              ;; ID dell'alert box
50 50 110 300                     ;; rettangolo di definizione
257                               ;; ID della item list
5555                              ;; i 4 stadi dell'alert
;; 5= 0101
;; 01 = beep n.1
;; 1  = l'alert va emesso
;; 0  = il primo elemento (OK) è il default


TYPE DITL
 ,257
 2
  BtnItem Enabled
  20 210 38 240
 OK

  StatText Disabled
  1 60 55 200
Record non Trovato
```

QuickDraw

Nelle sezioni precedenti si sono analizzati gli oggetti grafici su cui il programma opera, senza tuttavia tener conto delle basi matematico-geometriche su cui tali oggetti si fondano, e soprattutto senza curarci delle primitive di basso livello che ci permettono di veder apparire o scomparire sullo schermo finestre, menù, controlli, ecc.

Tutto l'impianto (strutture dati + routines) viene fornito da QuickDraw. Parleremo in questa sezione solamente della parte di questo insieme che viene direttamente utilizzata dal programma, rinviando per le altre strutture e routines di utilizzo più specificamente grafico al manuale *Inside Macintosh.*

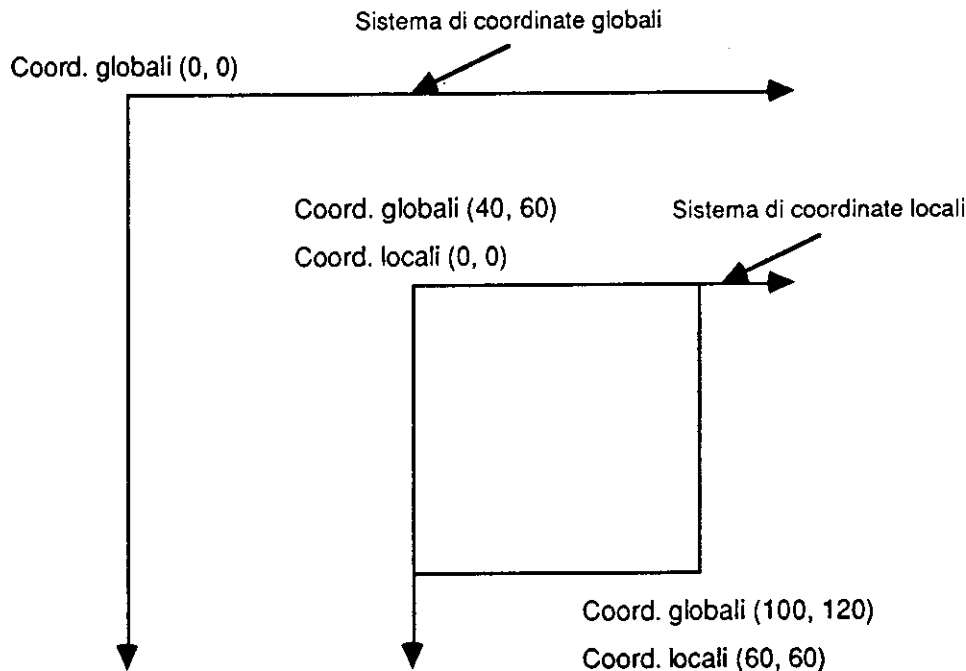
Le strutture dati più comunemente utilizzate rappresentano i concetti geometrici di punto (visto come una coppia di coordinate) e di rettangolo, definito dal punto in alto a sinistra (topLeft) e da quello in basso a destra (botRight). Ad essi si fa riferimento per molte routines grafiche incluse in QuickDraw.

Un rettangolo nel sistema di coordinate di QuickDraw

Le operazioni di disegno vengono effettuate sulle BitMap, che rappresentano i "fogli" su cui opera la penna di QuickDraw. Una BitMap è caratterizzata dal rettangolo di definizione e dalla zona di memoria in cui vengono "ricordati" i disegni sotto forma di bits (che rappresentano i pixel dello schermo). La BitMap più importante è ScreenBits, cioè quella che rappresenta tutto lo schermo di Mac e su cui vengono trasferiti i disegni che devono essere visualizzati: la zona di memoria ad essa associata è infatti il buffer di schermo, che viene periodicamente letto durante il refresh del video.

Per permettere al programmatore di avere a disposizione più BitMap virtuali, su cui compiere operazioni grafiche indipendenti, utilizzando su ognuna tipi diversi di penna, di pattern, di caratteri, ecc., QuickDraw mette a disposizione un oggetto più complesso, che è la vera e propria base dello sviluppo dell'interfaccia utente, la porta grafica. Ad ogni porta grafica vengono associate, tra le altre informazioni, una BitMap privata, i tipi e le caratteristiche della penna, del pattern e dei caratteri in uso (txfont in particolare contiene il numero della fonte e txsize l'altezza dei caratteri) e il rettangolo (portrect) all'interno del quale le operazioni vengono effettuate (che può essere più piccolo del rettangolo della BitMap). Ad ogni porta corrisponde poi un sistema di coordinate, con l'origine nell'angolo in alto a sinistra e le ascisse crescenti verso destra e le ordinate crescenti verso il basso.

Tutto l'insieme di QuickDraw deve essere inizializzato all'inizio del programma per mezzo della procedura *InitGraf*, cui va passato l'argomento @thePort, che rappresenta l'indirizzo delle locazioni di memoria in cui vengono conservate le informazioni globali di QD. Le routines di QuickDraw operano di norma sulla porta corrente, che deve quindi essere opportunamente aggiornata prima di ogni operazione grafica. A tale scopo esistono la funzione *GetPort*, che ritorna la porta corrente e la procedura *SetPort*, che determina la nuova porta corrente.

Il sistema di coordinate della porta può essere modificato, per mezzo della procedura *SetOrigin*(x,y), che trasla l'origine degli assi fino a che l'angolo in alto a sinistra dello schermo assume le coordinate (x,y). Poichè però il punto in cui si trova il mouse nel momento in cui si verifica un evento viene espresso in coordinate

assolute (dello schermo), occorre che, per i "conti" all'interno di una porta la cui origine sia stata modificata, tale punto venga trasformato in coordinate locali alla porta: è questo il compito della procedura *GlobaltoLocal*, cui corrisponde l'inversa *LocaltoGlobal*.

I sistemi globali e locali di coordinate

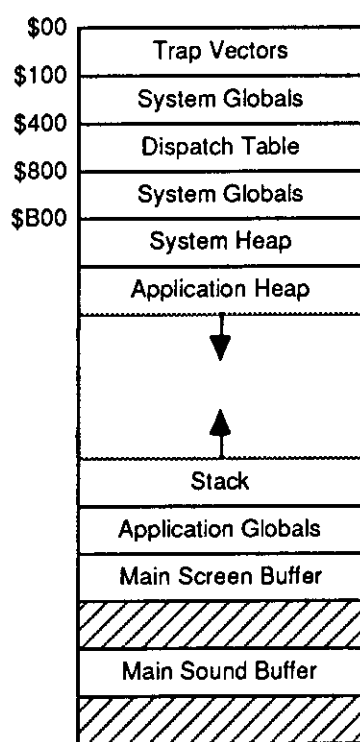
All'interno di QD sono poi disponibili le procedure per l'assegnamento delle coordinate a punti e rettangoli (*SetPt* e *SetRect*), per la verifica dell'inclusione di un punto in un rettangolo (*PtinRect*), per la modifica delle dimensioni di un rettangolo (*InsetRect* e *OffsetRect*), oltre alle routines per calcolare unione, intersezione di due rettangoli (*UnionRect* e *SectRect*) o l'uguaglianza tra due punti o due rettangoli (*EqualPt* e *EqualRect*).

Un'ultima procedura di QD che utilizziamo nel programma è *InvalRect*, che aggiunge un rettangolo alla lista delle zone del video che devono essere ridisegnate, facendo così in modo che venga ridisegnato correttamente durante l'esecuzione della routine relativa all'evento di aggiornamento (Update Event).

Come già premesso, non ci siamo occupati delle altre strutture dati e routines più specificamente grafiche, che permettono di generare e manipolare oggetti più complessi dei rettangoli, cioè gli ovali, i poligoni, le regioni e le pictures.

La gestione della memoria

La memoria di Macintosh è suddivisa in una zona di ROM (di 64Kbytes nella versione originale e di 128K nel Mac Plus), che contiene le routines del Toolbox, e in una più ampia zona di RAM, che varia, a seconda dei modelli, tra 128K e 1Mbyte.

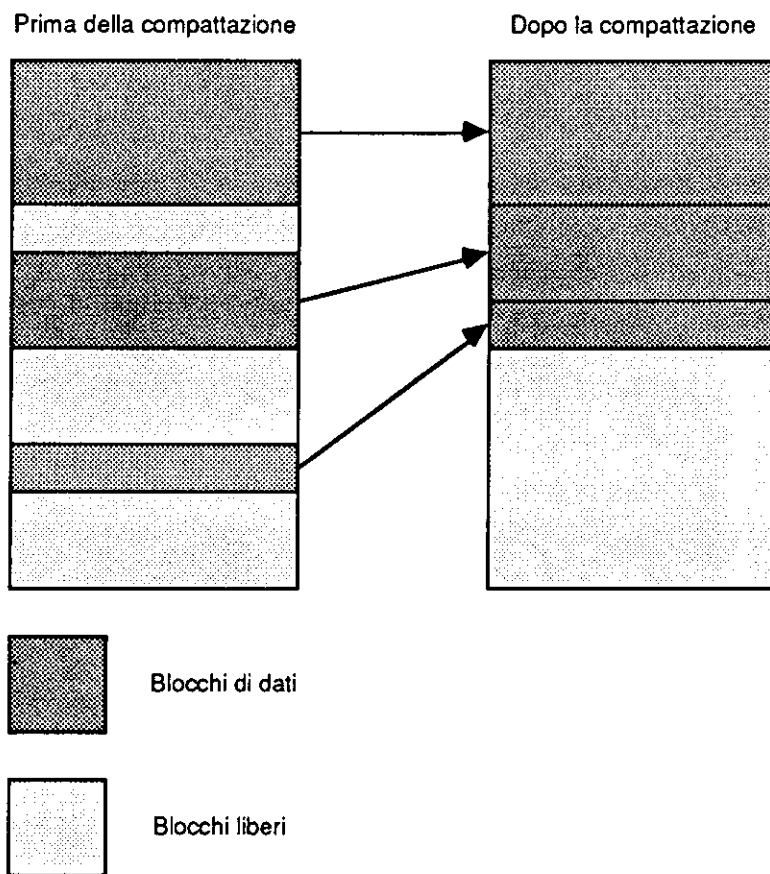
La RAM è a sua volta suddivisa in più zone, in parte utilizzate dal sistema, in parte lasciate libere per le applicazioni.
Gli indirizzi tra $0 e $FF contengono i vettori di trap, mentre tra $400 e $7FF sono conservati gli indirizzi delle routines di Toolbox. Le zone di memoria comprese tra $100 e $3FF e tra $800 e $AFF contengono le variabili globali del sistema. In fondo alla memoria (indirizzi più alti) sono allocati il buffer dello schermo e del suono.

Lo schema di allocazione della memoria di Macintosh

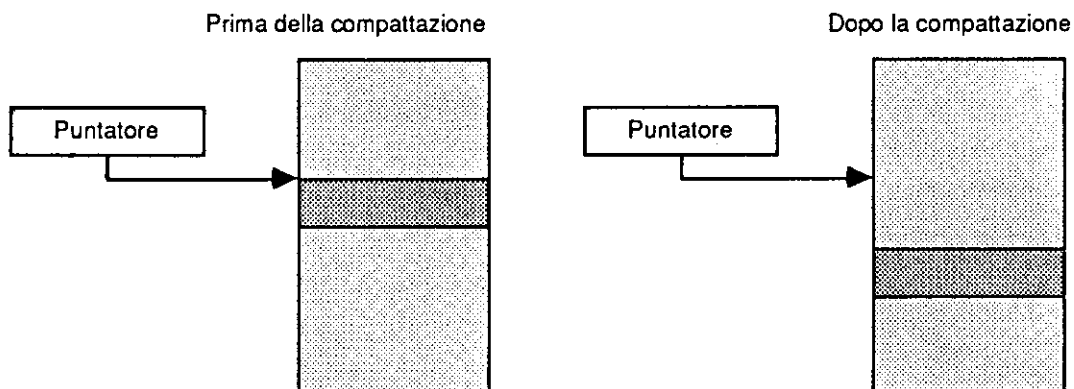
Nella memoria appena precedente questi buffer vi è lo spazio globale dell'applicazione, che contiene le variabili globali

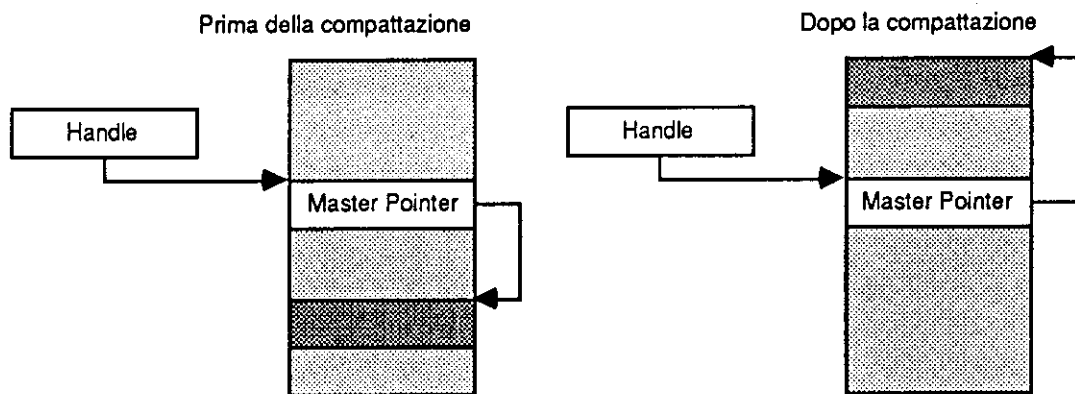
(comprese quelle di QuickDraw), i parametri dell'applicazione e la jump table (per la gestione degli indirizzi delle routines appartenenti ai vari segmenti rilocabili di codice).

Lo spazio disponibile per l'allocazione dinamica è compreso tra la fine dei globali di sistema ($B00) e l'inizio dei globali dell'applicazione. Quest'area è condivisa dalle due forme di allocazione: stack e heap. Lo stack "cresce" a partire dagli indirizzi più alti in memoria, a ridosso cioè dei globali dell'applicazione; viene utilizzato per il passaggio di parametri alle procedure, per tenervi gli indirizzi di ritorno e per le variabili locali. I linguaggi ad alto livello provvedono automaticamente alla gestione di tale zona.

Lo spazio di heap invece viene allocato e rilasciato solo e sempre su richiesta esplicita al sistema, in blocchi di qualsiasi dimensione. Lo heap è a sua volta suddiviso in due zone, una riservata al sistema, il cui contenuto non viene distrutto al termine di un programma utente, permettendo così che il sistema mantenga le sue strutture dati tra un programma e l'altro; l'altra è lo heap dell'applicazione, da cui vengono prelevati blocchi di memoria necessari ai programmi utente.

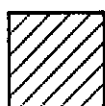
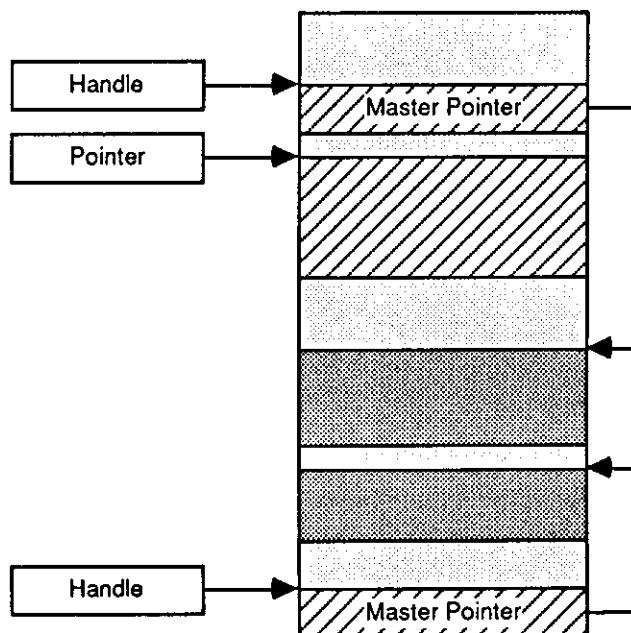
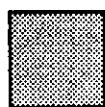
Dallo heap si possono prelevare blocchi di dimensione qualsiasi, e ad esso vanno rilasciati nel momento in cui non servono più, perchè possano essere riutilizzati. Durante l'esecuzione di un programma, un alto numero di allocazioni e deallocazioni tende a spezzettare la memoria in blocchi di dimensioni sempre minori, rendendo perciò a volte impossibile soddisfare richieste di blocchi nuovi. Quando ciò accade, il Toolbox cerca di recuperare lo spazio frammentato compattando i blocchi in uso, spostandoli uno vicino all'altro e raggruppando gli spazi inutilizzati in uno solo, da cui poi estrarre i blocchi richiesti. Ovviamente, se neanche dopo una compattazione della memoria è possibile soddisfare la richiesta, il Toolbox segnala l'errore di memory overflow.

La compattazione dello heap

Durante il processo di ricompattamento i blocchi cambiano il loro indirizzo originale e i puntatori ad essi non possono venire aggiornati, se il sistema non li conosce.

Il problema dei cosiddetti "dangling pointers"

Per risolvere questo problema, nel disegno del Toolbox è stata utilizzata una nuova entità, l'handle. Un handle per definizione è un puntatore ad un puntatore, che è unico e, soprattutto, non è rilocabile. Tale puntatore, detto Master Pointer, è conosciuto dal sistema ed è quindi facilmente aggiornabile. Se tutti i riferimenti a blocchi di memoria vengono fatti "passando" attraverso i Master Pointers, la loro consistenza è garantita.

Handle e Master Pointer

I blocchi accessibili tramite handles sono detti rilocabili, e vengono allocati per mezzo della funzione *NewHandle*, cui va passata la dimensione del blocco da allocare (sempre un numero pari, per carità!!!) e che ritorna un handle al nuovo blocco.

La dimensione di un blocco (cui si può accedere con la funzione *GetHandleSize*) non è statica: può essere diminuita o aumentata tramite *SetHandleSize*. Nel secondo caso il Toolbox sposta i dati contenuti nel vecchio blocco in nuovo blocco più grande, lasciando quindi le cose inalterate ad alto livello.

Per il rilascio di un blocco rilocabile viene utilizzata *DisposHandle*. E' possibile comunque allocare blocchi "alla vecchia maniera", tramite puntatori cioè, rendendo perciò questi blocchi non rilocabili. Le routines, simili alle precedenti, si chiamano *NewPtr*, *GetPtrSize*, *SetPtrSize* e *DisposPtr*.

Blocchi rilocabili e non rilocabili

Quando, all'interno del programma, si fa riferimento ad un record allocato in un blocco rilocabile, spesso conviene "accorciare" la strada verso i dati, utilizzando un puntatore temporaneo direttamente al blocco. In questo caso, bisogna che i riferimenti siano mantenuti consistenti per almeno il tempo di utilizzo del puntatore; occorre cioè congelare la situazione, rendendo temporaneamente non rilocabile il blocco in questione. Si usano a questo scopo le procedure *HLock* e *HUnLock* per "abbassare e alzare il termostato".

Durante il processo di compattamento, il Toolbox, nel tentativo di ricuperare memoria, cerca anche di rimuovere i blocchi rilocabili che siano dichiarati rilasciabili. All'atto della creazione ogni blocco è non rilasciabile, e il suo stato può essere modificato con *HPurge* e *HNoPurge*.

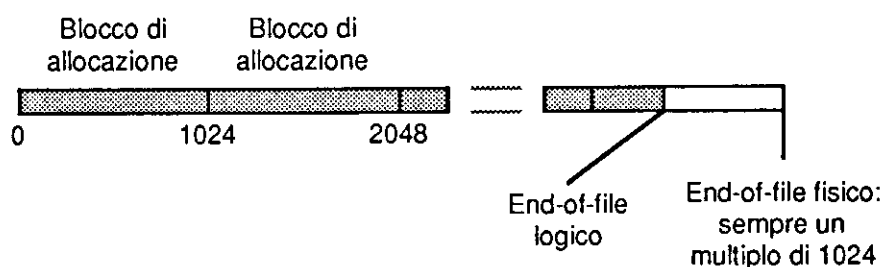
Nel nostro esempio, facciamo più volte uso delle routines per la manipolazione dei blocchi rilocabili. In particolare, la procedura che segue modifica la dimensione dei due blocchi che contengono la tabella dei record e la tabella degli indici per aggiungere un nuovo record.

```
PROCEDURE insert;

 BEGIN
  IF numrec = 0
   THEN
     .....    {riabilita i menù}

 numrec := numrec + 1;
 SetHandleSize(Handle(thetabhdl),numrec*SIZEOF(person));
 SetHandleSize(Handle(theindhdl),numrec*SIZEOF(integer));
 theindhdl^^[numrec - 1] := numrec - 1;

 { Scrivi i dati nel record di indice numrec - 1}
 ++++........
 END;
```

Esiste inoltre una serie di routines predefinite per la copia di blocchi rilocabili a blocchi non rilocabili, e tra blocchi omologhi. Nel nostro esempio però viene utilizzata una routine di livello più basso, *BlockMove*(fromptr, toptr, N), che trasferisce, senza controlli, N bytes a partire dall'indirizzo fromptr nella zona puntata da toptr.

La gestione dei files

Un file di Macintosh è un insieme di dati su memoria di massa, di cui sono noti la posizione fisica, la lunghezza (limitata solo dalle dimensioni del dispositivo di memorizzazione), il nome (fino a 64 caratteri), il tipo e il creatore, rappresentati da due stringhe di quattro caratteri, che servono al Finder per riconoscerne l'utilizzo (infatti un doppio click su un documento permette di lanciare l'applicazione che lo ha creato, passandole il nome del documento stesso). Nel nostro programma, il file su cui vengono conservati gli indirizzi ha come tipo 'ARCH' e creatore 'DEMO', mentre il programma è di tipo 'APPL' e creatore ancora 'DEMO', in modo che il Finder lo riconosca come l'applicazione che gestisce i files con lo stesso creatore.

I files sono visti dal sistema come sequenze di bytes, di lunghezza arbitraria, che vengono memorizzati in blocchi di allocazione di lunghezza fissa (diversa a seconda del tipo di dispositivo). Esistono perciò due posizioni di fine file (EndOfFile): fisica, che è la posizione dell'ultimo byte dell'ultimo blocco di allocazione e indica quindi quanti bytes sono riservati sul disco, e logica, che punta all'ultimo byte utile nel file e indica quanti byte di quelli allocati sono effettivamente significativi. Chiaramente l'EOF logico non può mai superare l'EOF fisico. *GetEOF* e *SetEOF* sono le routines utilizzare per leggere e modificare l'EOF logico. Da notare che se l'EOF logico modificato con *SetEOF* assume un valore maggiore di quello fisico, il file system automaticamente estende la lunghezza fisica del file.

End-of-file logico e fisico con blocchi di allocazione di 1K

Per indicare qual è la posizione corrente di utilizzo del file viene utilizzato il file mark. Per la sua manipolazione vengono fornite le due primitive *GetFPos* e *SetFPos*. Quest'ultima permette di spostare il mark rispetto all'inizio del file, alla sua posizione precedente o alla fine del file.

Per eseguire le operazioni di lettura e scrittura di files, è necessario innanzi tutto "aprirli" con la funzione *FSOpen*, che, dato il nome di un file e un numero di riferimento di volume, apre il file, ritorna come valore un codice di errore (NoErr, FNFErr, ecc.) e ritorna (in un parametro var) il numero di riferimento del file, che dovrà poi essere usato per riferirsi ad esso durante le operazioni successive.

Se invece il file non esiste ancora sul disco, occorre crearlo, con la funzione *Create*, e poi aprirlo, sempre con *FSOpen*, per ottenerne il numero di riferimento.

Una volta che il file è stato aperto, può essere letto o scritto con le due routines *FSRead* e *FSWrite*, cui passa il numero di riferimento, il numero di bytes da trasferire e un puntatore al buffer da utilizzare per il trasferimento dei dati. Le informazioni vengono lette o scritte sempre a partire dalla posizione corrente del mark del file.

L'operazione da eseguire al termine dell'utilizzo di un file è *FSClose*, che chiude il canale di comunicazione e rilascia la memoria utilizzata dal buffer.

Il Toolbox fornisce inoltre due routines (contenute nello Standard File Package) per le comunicazioni tra utente e programma che riguardino i files. Esse usano una struttura dati propria, il record di risposta (di tipo SFReply), attraverso il quale vengono trasmesse le informazioni necessarie. Le procedure sono *SFGetFile* e *SFPutFile*. La prima apre sullo schermo un dialog standard, in cui compaiono i nomi dei files presenti su disco (eventualmente solo quelli di alcuni tipi), i bottoni di **Apri**, **Annulla**, **Espelli** e **Unità Disco**. Il bottone **Apri** serve per restituire al programma che ha chiamato *SFGetFile* il nome del file selezionato (nel campo fname del record di risposta), **Annulla** cancella tutta l'operazione (ritornando False nel campo Good), **Espelli** permette di estrarre

il disco nel drive corrente e **Unità Disco** (attivo solo se nel drive esterno vi è un disco) cambia il drive da cui i files vengono letti.

SFPutFile emette invece un dialog, in cui viene richiesto all'utente un nome di un nuovo file, su cui andranno fatte le operazioni di scrittura. Nel dialog compaiono inoltre una stringa di commento all'operazione (per esempio: 'Salva con che nome?') ed un eventuale nome di file di default.

Nella procedura *Doopen*, che abbiamo scritto per rispondere alla richiesta dell'utente di aprire un file (con un click nell'item **Apri** del menù **File**), vediamo concretamente l'uso di queste routines.

```
PROCEDURE doopen;

  VAR
   thetypelist: SFTypeList;     { Lista di TIPI di files da
                                  cercare con SFGetFile }
   origin    : point;
   filename  : STR255;
   textlength: LONGINT;
   io        : OSErr;
   i         : INTEGER;

  BEGIN

   { Prepara le strutture interne per un nuovo documento }
   donew;

   { Se non è già noto il nome del file da aprire (passato
     dal finder) si chiede con il dialog standard }
   IF (reply.fName = '')
    THEN
     BEGIN
      origin.h := 50;
      origin.v := 50;
      thetypelist[0] := 'ARCH';          {Si possono aprire
                                     solo files di tipo
'ARCH'}
     {Richiedi il nome del file da leggere, mettendo il
      dialog nel punto "origin", cercando solo i files di
      1 tipo, contenuto in thetypelist, e ritorna il
      risultato dell'operazione nella variabile reply}
SFGetFile(origin, '', NIL, 1,thetypelist, NIL, reply);
     END;

   WITH reply DO
```

```
IF Good   {L'utente ha effettivamente scelto un file}
 THEN
  BEGIN
   {Aggiorna il titolo della finestra}
   SetWTitle(thewindow, fName);

   {Apri il file, leggi la posizione dell'EOF (cioè la
   lunghezza del file) e posiziona il mark all'inizio}
   io := FSOpen(fName, vRefNum, frefnum);
   {Controlla che non ci siano errori (segnalati da
    io<>0)}
   iocheck(io);
   io := GetEof(frefnum, textlength);
   iocheck(io);
   {Posizionati nel file denotato da frefnum alla
    posizione 0 relativa all'inizio del file (1)}
   io := SetFPos(frefnum, 1, 0);
   iocheck(io);

   { Dimensiona opportunamente la tabella degli
     indirizzi e degli indici }
   SetHandleSize(Handle(thetabhdl), textlength);
   numrec := textlength DIV SIZEOF(person);
SetHandleSize(Handle(theindhdl),numrec*SIZEOF(INTEGER));

   HLock(Handle(thetabhdl));
   HLock(Handle(theindhdl));

   { Leggi tutto il file direttamente nella tabella
     dei record e poi chiudilo }
   io := FSRead(frefnum, textlength, @thetabhdl^^);
   iocheck(io);
   io := FSClose(frefnum);
   iocheck(io);

   { Inizializza la tabella degli indici in ordine di
     lettura e poi riordina il tutto secondo il
     tipo di ordinamento corrente }
   FOR i := 0 TO numrec - 1 DO
    theindhdl^^[i] := i;
   sort;

   HUnLock(Handle(thetabhdl));
   HUnLock(Handle(theindhdl));

   { Abilita  opportunamente menù e singoli items }
   EnableItem(mymenus[FILEID], 3);
   EnableItem(mymenus[FILEID], 4);
   EnableItem(mymenus[FILEID], 5);
   EnableItem(mymenus[FILEID], 7);
```

```
        EnableItem(mymenus[EDITID], 0);
        EnableItem(mymenus[EDITID], 3);
        EnableItem(mymenus[EDITID], 4);
        EnableItem(mymenus[EDITID], 8);
        EnableItem(mymenus[EDITID], 9);
        EnableItem(mymenus[SORTID], 0);
        EnableItem(mymenus[SEARCHID], 0);
        DrawMenuBar;
        {seleziona il primo indirizzo}
        seltheline(0);
       END;
   END;
```

La procedura **Mysave** è il nucleo delle procedure **DoSave** e **DoSaveAs**, che la richiamano dopo aver sistemato gli argomenti nel record reply. Qui viene utilizzata la funzione *GetFInfo*, che dato il nome di un file e il numero di riferimento del volume su cui risiede, ritorna le informazioni relative in un parametro var di tipo FInfo (le stesse che si possono leggere selezionando l'item **Get Info** del menù **Archivio** del Finder) ed ha come valore un codice di errore (0 se il file esiste, FNFErr (File Not Found Error) altrimenti).
Al termine le informazioni relative al file vengono aggiornate sul disco, chiamando la procedura *FlushVol*, in modo da mantenere corretta la situazione anche in caso di improvvisi problemi, tipo mancanza di corrente, ecc.

```
PROCEDURE mysave;

 VAR
  theinfo   : FInfo;          { Record per info sui files }
  textlength: LONGINT;
  io        : OSErr;
  vRefNum   : INTEGER;
  datalength: LONGINT;
  i         : INTEGER;

 BEGIN

  { Viene usato reply o perchè MySave è stata chiamata
    da DoSaveAs che usa prima la SFPutFile o perchè è
    stata chiamata da DoSave per cui in reply c'è già il
    nome del file corrente }
  WITH reply DO
   IF Good
    THEN
```

```
BEGIN
 { Si reperiscono le informazioni sul file e, se non
   esiste, lo si crea. }
 io := GetFInfo(fName, vRefNum, theinfo);
 IF io = FNFErr                  {file not found}
  THEN
   BEGIN
    {Crea un file di nome fName, sul volume vRefNum,
     di creatore 'DEMO' e tipo 'ARCH'}
    io := Create(fName, vRefNum, 'DEMO', 'ARCH');
    {Controlla che tutto sia andato bene}
    iocheck(io);
   END;
 iocheck(io);

 { Apri il file }
 io := FSOpen(fName, vRefNum, frefnum);
 { Ora in frefnum c'è il numero di riferimento del
   file, da usare in tutte le operazioni successive
   per identificarlo }
 iocheck(io);
 { Posizionati al byte 0 a partire dall'inizio del
   file }
 io := SetFPos(frefnum, FSFromStart, 0);
 iocheck(io);

 { Congela il blocco che contiene la tabella degli
   indirizzi }
 HLock(Handle(thetabhdl));
 { Ogni record che verrà scritto è di lunghezza
   fissa }
 textlength := SIZEOF(person);

 { Dopo ogni scrittura di un record, datalength
   conterrà la dimensione effettiva del file
   aggiornato }
 datalength := 0;

 { Occorre scrivere un record per volta in modo da
   evitare di salvare quelli che erano stati
   "cancellati".
   Si cicla su tutti i record scrivendo solo quelli
   che hanno il campo "deleted" a FALSE, àggiornando
   contemporaneamente datalength. }
 FOR i := 0 TO numrec - 1 DO
  IF NOT thetabhdl^^[i].deleted
   THEN
    BEGIN
     { Scrivi sul file indicato da frefnum il testo
       di lunghezza textlength, che si trova a
       partire dall'i-esimo elemento della tabella }
```

```
        io:=FSWrite(frefnum, textlength, @thetabhdl^^[i]);
          iocheck(io);
          datalength := datalength + textlength;
         END;

      { Setta l'end-of-file logico in base a datalength e
        chiudi il file }
      io := SetEOF(frefnum, datalength);
      iocheck(io);
      io := FSClose(frefnum);
      iocheck(io);

      { Aggiorna il titolo della finestra nel caso fosse
        stata chiamata da DoSaveAs }
      SetWTitle(thewindow, fName);

      HUnLock(Handle(thetabhdl));

      { Completa effettivamente la scrittura sul disco }
      io := FlushVol(NIL, vRefNum);
      iocheck(io);
     END;
 END;
```

Nel caso in cui l'utente abbia creato una nuova tabella di indirizzi, o si voglia salvare una copia di quella corrente, sotto un altro nome, richiama l'item **Salva come** del menù **Archivio**. **Dosaveas** è la procedura che gestisce tale richiesta.

```
PROCEDURE dosaveas;

 VAR
  origin   : point;

 BEGIN
  IF winexist      { Solo se esiste una finestra aperta
                     c'è qualcosa da salvare }
   THEN
    BEGIN
     SetPt(origin, 100, 100);
     { Emetti il dialog standard (NIL) con l'angolo in
       alto a sinistra nel punto 100,100, la stringa di
       commento 'Salva con che nome ?', nessun nome di
       default e ricevi la risposta dell'utente nel
       record reply}
   SFPutFile(origin,'Salva con che nome ?','',NIL,reply);
     { Sistemati i parametri, puoi chiamare mysave }
     mysave;
```

```
    END;
  END;
```

Se l'utente seleziona invece l'item **Salva** del menù **Archivio**, intende aggiornare la copia su disco del documento corrente. Se questo non ha ancora un nome, la procedura **Dosave** lo chiede, passando il controllo alla procedura **DoSaveAs**, altrimenti chiama direttamente **MySave**. }

```
PROCEDURE dosave;

 BEGIN
  IF winexist
   THEN
    IF (reply.fName <> '')    { Reply contiene sempre il
                                nome del file corrente }
     THEN mysave
     ELSE dosaveas;
 END;
```

In tutte le routines viste finora, viene sovente fatto richiamo ad una procedura di controllo dei codici di errore ritornati dalle funzioni del file system, **iocheck**. Questa procedura, dato l'errore, emette un alert che indica quali problemi sono occorsi durante l'ultima operazione effettuata.

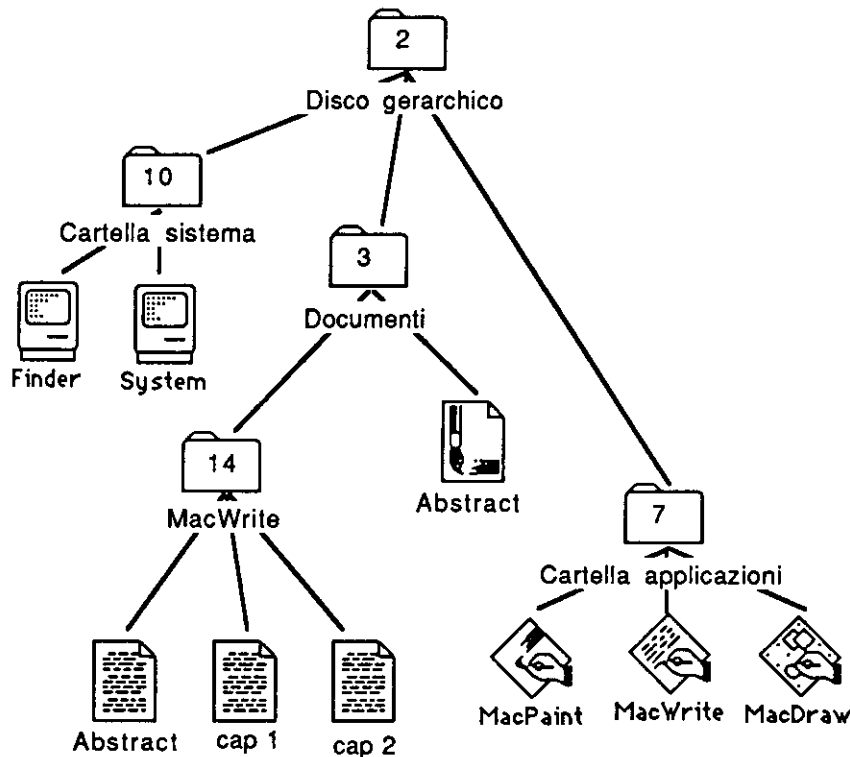
```
PROCEDURE iocheck(errNo: OSErr);

  VAR
   errstring: StringHandle;
   dummy    : INTEGER;

  BEGIN
   IF errNo <> NoErr    { Se si è verificato un errore }
    THEN
     BEGIN
      IF errNo < - 64
       THEN errNo := - 64;  { Gli errori inferiori a -64
                              sono tutti di basso livello}
      {leggi la stringa relativa all'errore nelle risorse}
      errstring := GetString(errNo);
      {Sostituiscila nell'alert al posto di ^0}
      ParamText(errstring^^, '', '', '');
      dummy := StopAlert(260, NIL);
     END;
```

```
END;
```

Il codice visto finora, scritto per essere utilizzato con il Macintosh File System (MFS), rimane valido anche in un ambiente in cui sia operativo il File System Gerarchico, o HFS, e il nuovo Finder che lo utilizza.

Sotto MFS, i folder o raccoglitori *simulavano* una schema a sottodirettrici (v. UNIX™, MS-DOS™, ecc.), ma solo all'interno del Finder, e a costo di un rallentamento nel tempo di esecuzione.

Sotto HFS, la scrivania rimane praticamente immutata, ma ora i folders sono vere sottodirettrici. In tal modo, non è più necesssario partizionare i dischi rigidi in più volumi per ridurre i tempi di accesso, ma i files possono essere sistemati in una struttura ad albero, in cui ogni file appartiene ad una direttrice, e più files con lo stesso nome possono coesistere su direttrici diverse.

HFS usa anche un nuovo modo di allocare lo spazio, cosicchè lo spazio minimo di allocazione è ora di 512 bytes, contro 1K per i floppy MFS o 2-4K per gli hard MFS. Il vantaggio sta nel maggior numero di piccoli files allocabili su disco.

Organizzazione del file system gerarchico

Il cambiamento maggiore dal punto di vista dell'utente si trova nei nuovi dialog per *SFGetFile* e *SFPutFile*.

Nella scroll box di *SFGetFile* vengono ora listati solo i files e le direttrici della direttrice corrente; per scendere nell'albero basta aprire il folder voluto, mentre per salire occorre clickare sul bottone in cui è scritta la direttrice corrente e scegliere dal menù pull-down ottenuto la nuova direttrice.

Nel dialog di *SFPutFile* vengono invece listati solo i folder, tra cui scegliere quello in cui andrà creato il file.

Dal punto di vista del programmatore, le cose, se sono stati seguiti attentamente i consigli sull'uso dei files ad alto livello, non dovrebbero cambiare di molto.

Ogni elemento dell'albero del file system, direttrice o file è detto nodo di catalogo (CNode). La direttrice alla base dell'albero è la

radice, ed ha lo stesso nome del volume su cui risiede. I nodi intermedi dell'albero sono sempre direttrici e i nodi terminali (foglie) sono files o direttrici vuote.

Ogni Cnode ha un nome. I nomi dei Cnode da attraversare per raggiungere un determinato nodo, concatenati e separati da ":" formano il *pathname*. Ogni elemento di un pathname, eccetto quello finale, deve essere una direttrice. Un pathname completo inizia con il nome della radice e descrive un percorso arbitrario lungo l'albero per raggiungere un determinato nodo. Un pathname parziale comincia con due punti, oppure consiste solo di un nome di nodo. Un pathname vuoto, cioè una coppia di due punti "::", equivale ad un passo indietro nell'albero, cioè a ritornare al nodo *padre* dell'attuale.

Internamente, ogni nodo direttrice ha un numero di identificazione, o DirID. I Cnodes possono essere indicati alle routines di HFS usando un pathname parziale ed un DirID, che denota la direttrice da cui parte il cammmino indicato dal pathname.

I pathname interi possono a volte essere più lunghi di 255 caratteri, e non sono quindi rappresentabili con stringhe Pascal. Il programmatore può però specificare una direttrice corrente con una chiamata a *OpenWD*, che ritorna un WDRefNum, o numero di riferimento della Working Directory, che può essere usata nelle successive chiamate al sistema.

I WDRefNum possono essere passati alle routines del file system al posto dei numeri di volume (vRefNum), senza creare problema alcuno. La direttrice corrente è una dei punti fondamentali per la compatibilità di HFS con MFS. Sotto HFS, *SFGetFile* ritorna un WDRefNum nel campo del record di risposta in cui, sotto MFS, ritorna il vRefNum.

Poichè le altre routines, come si è visto, riconoscono la differrenza e vi si adeguano senza problemi, il programmatore può ignorare il contenuto del record ritornato da *SFGetFile*. Il nostro programma, quindi, non facendo mai accessi diretti ai record di informazione relativi ai files, ma limitandosi ad agire con le routines di alto livello del file system, pur essendo stato creato

in un ambiente MFS, funziona correttamente anche sotto HFS.

Le stampe

Tutte le operazioni di stampa su Macintosh vengono effettuate per mezzo del Printing Manager, un insieme di routines in RAM e di tipi di dati, che permettono di usare le routines standard di QuickDraw per stampare testo e grafici su una stampante.

Il Printing Manager non è nelle ROM di Macintosh come molti altri Managers; per poterne utilizzare le routines è necessario collegare con il linker il codice dell'applicazione con quello del Printing Manager fornito assieme al sistema di sviluppo.

Il Printing Manager è disegnato in modo che l'applicazione non si debba preoccupare del tipo di stampante collegata a Macintosh; si possono chiamare le stesse routines indipendentemente dalla stampante. L'indipendenza dalla stampante è possibile perchè il codice effettivo di gestione della stampante (differente per ogni modello) è contenuto in un file separato di risorse sul disco utente. Questo file contiene un device driver chiamato **Printer Driver**, che gestisce le comunicazioni tra il Printing Manager e la stampante.

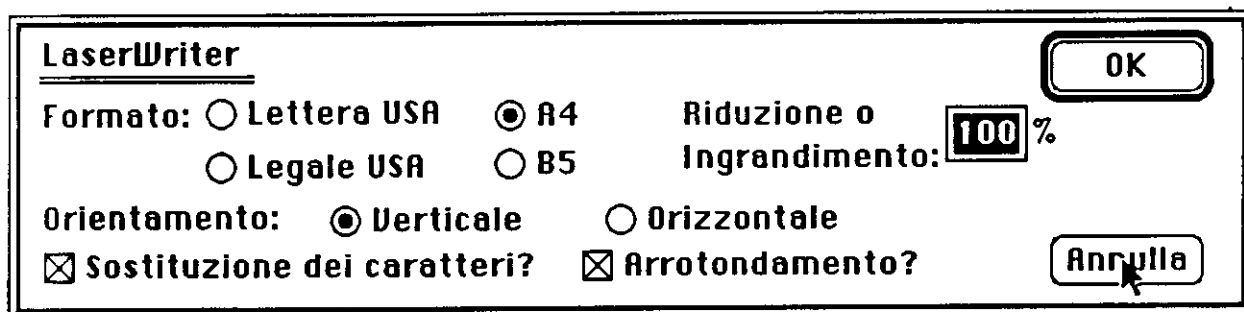
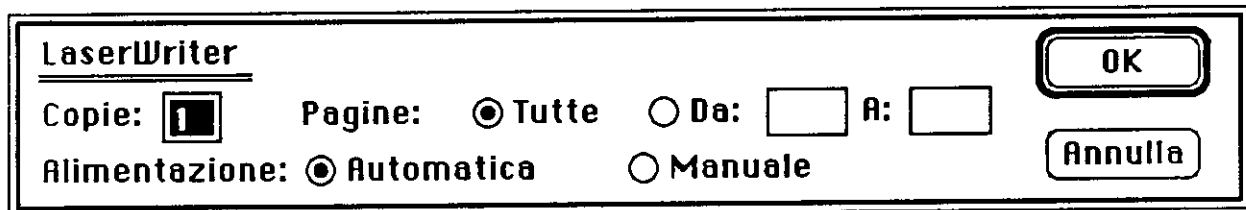
L'utente installa una nuova stampante per mezzo dell'accessorio **Scelta stampante** che fornisce al Printing Manager un nuovo file di risorse per la stampa. Questo processo è trasparente per l'applicazione e l'applicazione non deve fare nessuna assunzione sul tipo di stampante che verrà utilizzata.

Il programma definisce l'immagine da stampare usando una grafPort di stampa, che è una grafPort di QuickDraw con l'aggiunta di alcuni campi per adattarla alle operazioni di stampa. Il Printing Manager fornisce una grafPort di stampa quando si apre un documento per stamparlo. Quindi si stampano il testo e i grafici disegnandoli in questa porta grafica con le routines di QuickDraw, come se si stesse disegnando sullo schermo. Il Printing Manager installa le sue versioni particolari delle routines di disegno di basso livello nella grafPort di stampa, facendo sì che le chiamate di alto livello di QuickDraw gestiscano la stampante invece di disegnare sullo schermo.

Per usare il Printing Manager è necessario inizializzare prima

QuickDraw, il Font Manager, il Window Manager, il Menu Manager, TextEdit e il Dialog Manager. La prima routine del Printing Manager da chiamare è *PrOpen* e l'ultima *PrClose*. Prima di poter stampare c'è bisogno di un print record valido per la stampante in uso. Si può usare un print record esistente (ad es. salvato insieme al documento) o inizializzarne uno nuovo chiamando *PrintDefault* e *PrintValidate*. Per creare un nuovo print record si deve prima creare un handle ad esso con la routine *NewHandle* del Memory Manager. Nel nostro programma queste operazioni sono effettuate una volta per tutte nella fase di inizializzazione.

La configurazione (o set-up) delle stampe viene ottenuta richiamando con la routine *PrStlDialog* il dialog standard per la stampante prescelta. Questa routine, dato un handle, ritorna nel blocco puntato le informazioni ricevute. Con questo dialog (Style Dialog), che deve essere presentato all'utente quando questo sceglie l'opzione **Formato di stampa** dal menù **Archivio**, è possibile specificare ogni opzione che modifica le dimensioni della pagina di stampa. Nella figura si vede il dialog per la LaserWriter.

LaserWriter
OK
Formato: ○ Lettera USA ◉ A4 Riduzione o Ingrandimento: 100 %
○ Legale USA ○ B5
Orientamento: ◉ Verticale ○ Orizzontale
☒ Sostituzione dei caratteri? ☒ Arrotondamento?
Annulla

Il dialog per la scelta dello stile di stampa

```
PROCEDURE dopsetup;

 VAR
  ok        : Boolean;

 BEGIN
  ok := PrStlDialog(printhdl);
 END;
```

Si deve invece presentare il dialog di lavoro (job dialog) quando l'utente sceglie di stampare con l'opzione **Stampa** del menù **Archivio**. Per mostrare questo dialog si chiama la routine *PrJobDialog* del printing manager, cui si passa sempre lo handle al record di stampa in cui verranno memorizzate le scelte fatte dall'utente nel dialog. Il dialog richiede informazioni su come stampare il documento questa volta , come il tipo di alimentazione dei fogli, il range di pagine da stampare e il numero di copie. nella figura successiva vediamo il job dialog standard della LaserWriter.

LaserWriter OK
Copie: 1 Pagine: ◉ Tutte ○ Da: [ ] A: [ ]
Alimentazione: ◉ Automatica ○ Manuale Annulla

Il job dialog standard della LaserWriter

Ci sono due metodi fondamentali di stampa: draft e spool. Stampando in modo draft le chiamate a QuickDraw vengono direttamente convertite in codici di controllo che la stampante è in grado di interpretare e che vengono immediatamente usati per gestire la stampante.

La stampa in modo spool è un processo in due fasi. Prima il Printing Manager scrive una rappresentazione dell'immagine del documento da stampare su un file o in memoria (fase di "spooling"). Quindi questa informazione viene convertita in una bit image e stampata (fase di stampa).

Le fasi di spooling e di stampa sono state separate per poter ottimizzare l'utilizzo della memoria. La fase di spooling richiede solo 3K di memoria circa, ma può aver bisogno di larghe parti del codice e dei dati dell'applicazione in memoria. La fase di stampa invece richiede dai 20K ai 40K per il codice di stampa, i buffers e le fonti, ma la maggior parte del codice e dei dati dell'applicazione non sono più necessari. Così per ottimizzare la gestione della memoria si può creare un segmento separato per il codice di stampa, in modo da poter liberare la memoria dagli altri segmenti

e dai dati durante la fase di stampa, per poi ricaricarli in memoria quando la stampa è terminata.

In generale per stampare un documento si utilizzano le seguenti procedure del Printing Manager:

- *PrOpenDoc*, che restituisce una grafPort di stampa inizializzata per la stampa in spool o in draft (a seconda del campo bJDocLoop del print record).
- *PrOpenPage*, che inizia ogni nuova pagina (riinizializzando la grafPort).
- Le routines di QuickDraw, per disegnare la pagina nella grafPort aperta con *PrOpenDoc*.
- *PrClosePage*, che termina la pagina.
- *PrCloseDoc*, alla fine dell'intero documento, per chiudere la grafPort di stampa.

Ogni pagina viene stampata immediatamente (stampa draft) oppure viene memorizzata su disco o in memoria (stampa in spool). Se è stata scelta la stampa in spool, dopo la fase di spool (che finisce con la chiamata a *PrCloseDoc*) è necessario iniziare la fase di stampa chiamando *PrPicFile*.

La procedura che realizza la stampa degli indirizzi nel programma di esempio è **doprint**. La struttura della procedura è la seguente:

- Una fase di inizializzazione, in cui si richiama il job dialog,si salva la porta corrente in una variabile, si apre la porta di stampa, si fissano la fonte e le dimensioni dei caratteri, si definiscono i rettangoli in cui verranno disegnati i campi dei record, si calcolano il numero di record che verranno stampati in una pagina e infine il numero di pagine da stampare.
- Un ciclo sul numero di copie del documento da stampare, nel caso si utilizzi la stampa in modo draft (per la stampa in spool il numero di copie viene gestito da *PrPicFile*).

- Un ciclo sul numero di pagine da stampare, in cui si apre la pagina di stampa con *PrOpenPage*, si stampa l'intestazione del documento (nome del file).
  - Un ciclo sul numero di record per pagina in cui si disegnano effettivamente i campi dei record e si spostano i rettangoli in cui verranno disegnati i successivi.
  - Dopo aver disegnato tutti i record di una pagina si riportano i rettangoli di destinazione ai valori di default e si chiude la pagina con *PrClosePage*.
- Dopo aver terminato i cicli sul numero di pagine e di copie si chiude la porta di stampa con *PrCloseDoc* e se è il caso (stampa in spool) si inizia la fase di stampa con *PrPicFile* e infine si resetta la porta corrente al valore che aveva prima delle operazioni di stampa.

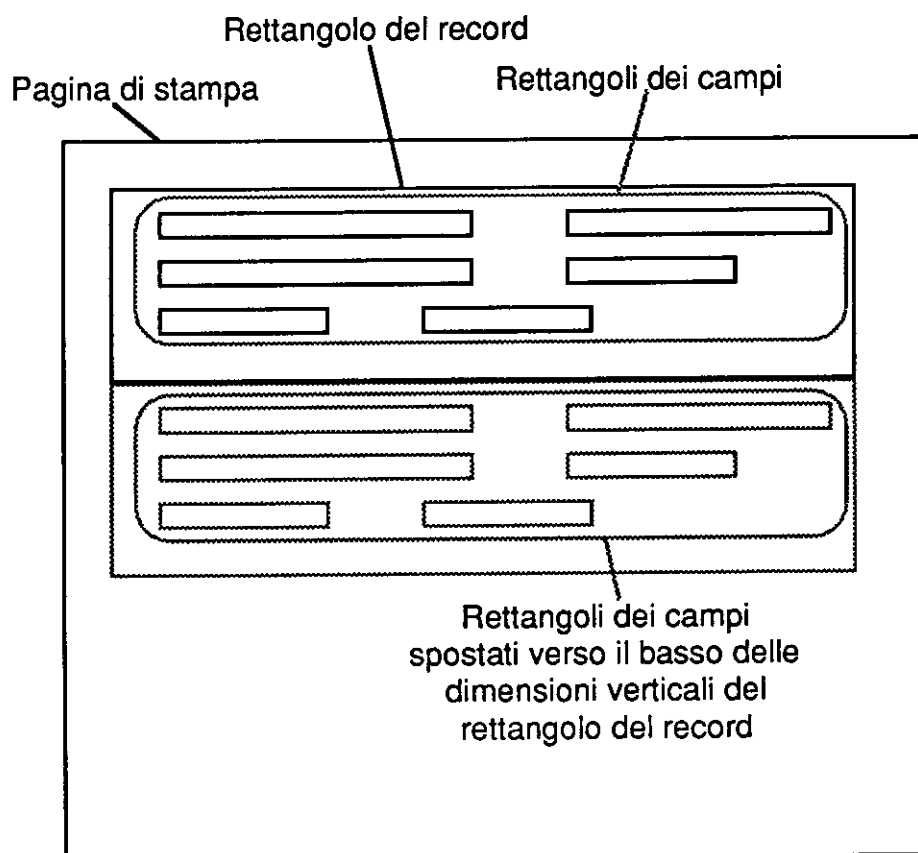
```
PROCEDURE doprint;

 VAR
  ok        : Boolean;
  prport    : TPPrPort;
  status    : TPrStatus;
  headrect, recrect, nomerect, cogrect, cittarect,
  viarect, caprect, telrect: Rect;
  i, j, k, nrectperpage, npages, loop, curind: INTEGER;
  astr      : STR255;

 { Assegna i valori ai rettangoli in cui verranno scritti
   i campi dei record.
   Per rendere più flessibile la stampa si dovrebbe
   utilizzare una procedura che permetta all'utente di
   definire interattivamente questi rettangoli. }

 PROCEDURE defrect;

  BEGIN
   SetRect(headrect, 10, 10, 300, 20); { Intestazione }
   SetRect(recrect, 0, 0, 300, 140);   { Spazio per un
                                         record }
   SetRect(nomerect, 20, 20, 120, 40); { Nome }
   SetRect(cogrect, 140, 20, 300, 40); { Cognome }
   SetRect(cittarect, 20, 60, 120, 80); { Citta' }
```

```
  SetRect(viarect, 140, 60, 200, 80); { Via }
  SetRect(caprect, 220, 60, 280, 80); { CAP }
  SetRect(telrect, 20, 100, 100, 120); { Telefono }
 END;

BEGIN

{ Richiamo il dialog standard per la definizione dei
  parametri di questa operazione di stampa }
 ok := PrJobDialog(printhdl);
 IF ok
  THEN
   BEGIN        { L'utente ha premuto OK nel dialog }
    GetPort(saveport);
    prport := PrOpenDoc(printhdl, NIL, NIL);

    { Definisco fonte e dimensione dei caratteri con cui
      scrivo sulla porta di stampa (quelle di default
      possono essere diverse), quindi disegno un blank
      per far caricare la fonte e la blocco in memoria }
    TextFont(MYTXFONT);
    TextSize(MYTXSIZE);
    drawChar(' ');
    SetFontLock(TRUE);

    defrect;

    { Calcolo il numero di record che stanno in una
      pagina }
    WITH printhdl^^.PrInfo DO
 nrectperpage :=(RPage.bottom - RPage.top - recrect.top)
                 DIV (recrect.bottom - recrect.top);

    { Calcola in numero di pagine da stampare }
    npages := ((numrec - 1) DIV nrectperpage) + 1;
    IF printhdl^^.prJob.BJDocLoop = BSpoolLoop
     THEN loop := 1
     ELSE loop := printhdl^^.prJob.iCopies;
    FOR i := 1 TO loop DO
     BEGIN
      curind := 0;
      FOR j := 1 TO npages DO
       BEGIN
        PrOpenPage(prport, NIL);

        { Definizione e stampa dell'intestazione }
        IF reply.fName = ''
         THEN astr := 'ARCHIVIO: Senza Nome'
         ELSE
          BEGIN
           astr := 'ARCHIVIO: ';
```

```
         astr := Concat(astr, reply.fName);
        END;
      TextBox(POINTER(ORD4(@astr) + 1), Length(astr),
              headrect, teJustLeft);

      HLock(Handle(thetabhdl));
      HLock(Handle(theindhdl));
      k := 1;

      { Ciclo per la stampa dei record su una pagina }
    WHILE (k <= nrectperpage) AND (curind < numrec) DO
       WITH thetabhdl^^[theindhdl^^[curind]] DO
        BEGIN
      TextBox(POINTER(ORD4(@nome) + 1),
              SIZEOF(nome) - 1, nomerect, teJustLeft);
      TextBox(POINTER(ORD4(@cognome) + 1),
            SIZEOF(cognome) - 1, cogrect, teJustLeft);
    {Eccetera...}
    { Dopo aver stampato un record  sposto i
      rettangoli più in basso }
    OffsetRect(nomerect,0,recrect.bottom-recrect.top);
    {Eccetera...}

         k := k + 1;
         curind := curind + 1;
        END;
    { Resetto i rettangoli ai valori iniziali per una
      nuova pagina }
      defrect;

      HUnLock(Handle(thetabhdl));
      HUnLock(Handle(theindhdl));
      PrClosePage(prport);
     END;                            {j loop }
   END;                              {i loop }

  IF printhdl^^.prJob.BJDocLoop = BSpoolLoop
   THEN PRPicFile(printhdl, NIL, NIL, NIL, status);
  PrCloseDoc(prport);
  SetPort(saveport);
 END;
END;
```

Per le operazioni di disegno delle stringhe relative ai campi dei record si è utilizzata la procedura *TextBox* cui si passano come parametri l'indirizzo e la lunghezza della stringa da disegnare, il rettangolo in cui disegnarla e il tipo di giustificazione.

Per il posizionamento delle scritte sul foglio si utilizza un

rettangolo per ogni campo del record e un rettangolo per ogni record. I rettangoli dei campi sono contenuti nel rettangolo del record e vengono definiti inizialmente con la procedura locale **DefRect**. Ogni volta che si cambia record sulla stessa pagina i rettangoli dei campi vengono spostati verso il basso di un numero di pixels pari all'altezza del rettangolo del record.

Utilizzo dei rettangoli per la stampa dei campi

Taglia copia e incolla

Uno dei punti cardine dell'interfaccia Macintosh è costituito dalla possibilità di comunicare dati tra applicazioni diverse. A tale scopo viene utilizzata una particolare struttura dati, lo Scrap, in cui vengono mantenute le informazioni "tagliate" o "copiate" da un programma, che da un altro possono essere "incollate" e riutilizzate.

Nello Scrap va tenuto un solo elemento, l'ultimo copiato o tagliato, ma in verità se ne possono mantenere più copie, di tipo differente. Si può cioè registrare lo stesso dato sotto diverse rappresentazioni, per permettere a varie applicazioni di scegliere quella più adatta alle proprie esigenze.

Per ottenere il contenuto dello Scrap facciamo uso della funzione *GetScrap*; essa accetta come parametri un handle al buffer in cui lo scrap va copiato (che quindi deve già essere stato allocato tramite una *NewHandle*), il tipo di risorsa voluto e un var in cui ritorna il numero di bytes letti; se l'handle passato è NIL, *GetScrap* non copia i dati, ma ne verifica solo la presenza e la lunghezza. Un valore negativo segnala un errore, mentre se è positivo significa che *Getscrap* ha trovato dei dati e ne ritorna la lunghezza.

Al contrario, per trasferire un elemento utilizziamo *PutScrap*, che appende il nuovo oggetto al vecchio contenuto. Vuole come parametri la lunghezza, il tipo e un puntatore all'elemento da inserire e restituisce un codice d'errore.
Per far sì che lo Scrap venga pulito prima del trasferimento occorre richiamare la funzione *ZeroScrap*.

Lo scrap risiede normalmente nello heap dell'applicazione, ma se, per ragioni di spazio, occorre "scaricarlo" su disco, si può richiamare *UnLoadScrap*, che ne trasferisce il contenuto sul ClipboardFile. La funzione opposta è *LoadScrap*, che va richiamata generalmente all'inizio di una applicazione o alla prima selezione del comando di Paste.

Nel nostro esempio abbiamo previsto l'utilizzo dello Scrap nella versione 'TEXT', ridefinendo in parte il comportamento delle

tradizionali Cut Copy e Paste, per adattarlo al tipo particolare di dati manipolati.

Copy trasferisce un indirizzo completo nello Scrap, Cut lo trasferisce e marca "cancellato" il record e infine Paste tenta di utilizzare il contenuto dello Scrap, interpretandolo come un insieme di sei stringhe (nome, cognome, via, città, CAP e telefono), separate da TAB o Return.
In queste procedure vengono utilizzate due routines, **convfromscrap** e **convtoscrap**, che verranno illustrate nel dettaglio in una sezione successiva.

```
PROCEDURE mycopy;  {copia nello scrap il record corrente}

  VAR
   along, len: LONGINT;
   astr      : STR255;

  BEGIN
   along := ZeroScrap;                       { Pulisce lo scrap }
   convtoscrap(astr);              { Converte in formato TEXT }
   len := Length(astr);
   { Inserisce nello scrap }
   along := PutScrap(len, 'TEXT', POINTER(ORD4(@astr)+1));
  END;


 PROCEDURE mycut;
   { Copia nello scrap il record corrente, lo marca come
     cancellato e aggiorna la finestra }

  BEGIN
   mycopy;
   {marca il record "cancellato"}
   WITH thetabhdl^^[theindhdl^^[currecind]] DO
    deleted := NOT deleted;
   {i dati sono stati modificati}
   dirty := TRUE;
   {aggiorna il contenuto della finestra}
   fillwindow;
  END;
```

```
PROCEDURE mypaste;
{ Preleva dallo scrap un oggetto di tipo TEXT;
  se questo è compatibile con il formato interno lo
  converte e inserisce il record corrispondente
  aggiornando la finestra, altrimenti visualizza un alert
  per segnalare l'incompatibilità }


 VAR
  along,          { Codice di errore dello scrap manager o
                    lunghezza dell'oggetto di tipo TEXT }
  offset     : LONGINT;
  astr       : STR255;
  ahdl       : Handle;
               { Handle all'oggetto caricato dallo scrap }
  itemhit    : INTEGER; { Campo dell'alert selezionato }

 BEGIN
  along := GetScrap(NIL, 'TEXT', offset);
   { Vede se c'è un oggetto di tipo TEXT }

  { Se c'è ed è piu corto di 255 caratteri converti,
    inserisci e aggiorna la finestra }
  IF (along > 0) AND (along < 255)
   THEN
    BEGIN

  {Se è il primo di una nuova finestra riabilita i menù }
     IF numrec = 0
         THEN
          ......
          ......

     { Alloca un Handle per l'oggetto }
     ahdl := NewHandle(0);
     { Copia l'oggetto dallo scrap }
     along := GetScrap(ahdl, 'TEXT', offset);
     HLock(ahdl);
     { Aggiorno la lunghezza della stringa }
     astr[0] := CHR(along);
     BlockMove(@ahdl^^, POINTER(ORD4(@astr) + 1), along);
     { Copio l'oggetto nella stringa }
     HUnLock(ahdl);
     DisposHandle(ahdl);

     { Inserisco il nuovo record e aggiorno la finestra }
     numrec := numrec + 1;
 SetHandleSize(Handle(theindhdl),numrec*SIZEOF(INTEGER));
 SetHandleSize(Handle(thetabhdl),numrec*SIZEOF(person));
     theindhdl^^[numrec - 1] := numrec - 1;
```

```
      convfromscrap(astr, thetabhdl^^[numrec - 1]);

      sort;
     END

    { Se il formato non è compatibile avvisa l'utente con
      un alert }
    ELSE itemhit := StopAlert(257, NIL);

  END;
```

Gli ordinamenti e altre utilities

La procedura che segue viene utilizzata per allineare i record alla loro lunghezza di definizione, in modo da favorire le operazioni di stampa. Viene qui fatto un uso in realtà non elegantissimo del formato delle stringhe Pascal, che sono viste come array di dimensione variabile, il cui primo elemento (0) contiene la lunghezza della stringa.

```
PROCEDURE blankfill(addr: Stringptr; dim: INTEGER);

 VAR
  i, len    : INTEGER;

 BEGIN
  len := ORD(addr^[0]);{leggi la lunghezza della stringa}

  FOR i := len + 1 TO dim - 1 DO
   addr^[i] := ' '; {aggiungi gli spazi mancanti}

  addr^[0] := CHR(dim - 1);{infine aggiorna la lunghezza}
 END;
```

La procedura di ordinamento, che viene descritta più sotto, riceve come parametri anche due funzioni, una per il confronto tra elementi e l'altra per accedervi, allo scopo di renderla il più generica possibile. Le due funzioni vengono descritte qui di seguito.

Compare, dati due puntatori generici, ritorna True se il primo punta ad un oggetto "minore" del secondo, False in caso contrario. Il confronto viene effettuato tenendo conto del campo rispetto al quale l'ordinamento va fatto (cursort).

```
FUNCTION compare(Ptr1, Ptr2: UNIV ptr): Boolean;

 VAR
  strp1, strp2: Stringptr;
  longp1, longp2: longptr;
  i          : INTEGER;

 BEGIN
{ Per tutti i campi non numerici si usa la routine
```

```
  standard IUCompString }
   IF cursort <> SORTDATA
    THEN
     BEGIN
{ Si devono passare 2 stringhe, quindi convertiamo i ptr.}
      strp1 := Stringptr(Ptr1);
      strp2 := Stringptr(Ptr2);
      i := IUCompString(strp1^, strp2^);
      compare := i < 0;
     END
    ELSE
     BEGIN
      longp1 := longptr(Ptr1);
      longp2 := longptr(Ptr2);

      {le date invadono anche il bit di segno dei LONGINT;
       con uno shift a destra di un bit si passa a una
       risoluzione di due secondi, ma si elimina il
       problema del segno.}
      compare := BitShift(longp1^, - 1) <
                 BitShift(longp2^, - 1);
     END;
  END;
```

Risolto il problema del confronto, occorre definire una funzione che ritorni un puntatore al campo del record corrispondente all'ordinamento corrente. Il parametro passato ad **access** è un puntatore all'elemento della tabella degli indici in cui è contenuto l'indice nella tabella dei record dell'elemento cui si vuole accedere.

```
FUNCTION access(aPtr: UNIV ptr): ptr;

 VAR
  ip       : intptr;

 BEGIN
  ip := intptr(aPtr);

  { In base al tipo di ordinamento corrente ritorna il
    puntatore al campo opportuno }
  CASE cursort OF
   SORTNOME: access := @thetabhdl^^[ip^].nome;
   SORTCOGN: access := @thetabhdl^^[ip^].cognome;
   SORTCITTA: access := @thetabhdl^^[ip^].citta;
   SORTVIA: access := @thetabhdl^^[ip^].via;
   SORTCAP: access := @thetabhdl^^[ip^].cap;
```

```
    SORTTEL: access := @thetabhdl^^[ip^].tel;
    SORTDATA: access := @thetabhdl^^[ip^].data;
   END;
  END;
```

La procedura di ordinamento sotto descritta, **sort**, ha il compito di congelare la situazione dei blocchi rilocabili durante il riordino della tabèlla degli indici, che viene fatto per mezzo di **QuickSort**, procedura cui vengono passati il puntatore alla tabella da ordinare, i due limiti inferiore e superiore della tabella, le dimensioni di ogni elemento e le due funzioni, di confronto e di accesso. Al termine rende di nuovo rilocabili i blocchi e aggiorna la finestra degli indirizzi.

```
PROCEDURE sort;

 BEGIN

   { Usa QuickSort. Il tipo di ordinamento da usare viene
     deciso dalla funzione "Access" in base alla
     variabile "CurSort" }
  HLock(Handle(thetabhdl));
  HLock(Handle(theindhdl));
  QuickSort(theindhdl^, 0, numrec - 1, SIZEOF(INTEGER),
            compare, access);
  HUnlock(Handle(thetabhdl));
  HUnlock(Handle(theindhdl));

  { Aggiorna la finestra }
  fillwindow;
  adjustscrollbars;
 END;
```

Vediamo ora in breve, la procedura di ordinamento **Quicksort**. Essa accetta sei parametri:

- **arrayToSort** è il puntatore all'array di record da ordinare;
- **lo** e **hi** sono gli indici iniziale e finale dell'array di cui il primo elemento ha indice 0;
- **DimOfRec** è la dimensione del record in bytes;

- **comp** è la funzione di confronto, i cui due argomenti sono i puntatori agli elementi da confrontare, e restituisce TRUE se gli elementi sono ordinati e FALSE altrimenti;

- **extr** è la funzione di estrazione: l'argomento è il puntatore a un elemento dell'array e restituisce il puntatore al campo di confronto (chiave di ordinamento).

L'array da ordinare deve essere non rilocabile, quindi se si usa un handle è necessario chiamare hlock prima di eseguirla.
La procedura è praticamente dummy, il vero lavoro viene eseguito dalla procedura **QuickS**.

QuickSort serve solo per rendere visibili alle procedure in essa contenute quei parametri che rimarranno invariati durante l'ordinamento, senza appesantire la ricorsione.

```
PROCEDURE QuickSort(arrayToSort : UNIV Ptr;
                    lo, hi, DimOfRec : integer;
      Function comp(vectItem, pivot : UNIV Ptr)  : Boolean;
      Function extr(recPtr : UNIV Ptr) : ptr);


  VAR
        Temp         : Ptr;

  BEGIN
    Temp := NewPtr(DimOfRec);
    QuickS(lo, hi);
    DisposPtr(temp);
  END;
```

L'algoritmo di QuickSort qui implementato può essere informalmente descritto come segue.

Dato un array disordinato, se ne sceglie un elemento a caso (ad es. il primo) detto Pivot e lo si mette in quella che sarà la sua posizione definitiva, facendo sì che alla sua "sinistra" ci siano solo elementi "minori" e alla sua destra solo elementi "maggiori" secondo il criterio di ordinamento (Questa operazione viene svolta dalla funzione **FindPosOfPivot**). Quindi si ordinano con lo stesso metodo i due sottoArray formati dagli elementi a sinistra e a destra del Pivot. Il procedimento termina quando l'Array da

ordinare è composto da un solo elemento.

```
PROCEDURE QuickS(lo, hi : integer);

  VAR
    PivotPos    : integer;

  BEGIN {Quicks}
    IF (lo < hi)
      THEN
        BEGIN
          {Cerca la posizione del pivot}
          PivotPos := FindPosOfPivot(lo, hi);
          {Ordina la parte a sinistra del Pivot}
          QuickS(lo, PivotPos - 1);
          {Ordina la parte a destra del Pivot}
          QuickS(PivotPos + 1, hi);
        END;
  END; {QuickS}
```

La funzione **FindPosofPivot** è il nucleo dell'ordinamento. Prende l'elemento di indice "lo" (Pivot), lo inserisce nella giusta posizione nel sottoArray fra gli indici "lo" e "hi" e restituisce la posizione finale del Pivot.

I due indici i e j vengono spostati dalle estremità dell'Array verso il centro finchè non si trovano elementi "fuori posto", che vengono scambiati fra di loro; il ciclo termina quando i e j si incontrano e questa è la posizione definitiva del Pivot.}

```
FUNCTION FindPosOfPivot(lo, hi: integer) : integer;

  VAR
    Pivot     : Ptr;
    i, j      : integer;

  BEGIN
    {Parti dal primo}
    Pivot := ArrPtr(lo);
    i := lo;
    j := hi;
    {Per tutti gli elementi compresi tra i posti i e j}
    WHILE (i < j) DO
      BEGIN
```

```
        {spostati da sinistra verso il centro cercando
         un elemento non ordinato rispetto al Pivot}
        WHILE (i < hi) AND
           NOT (comp(extr(pivot), extr(arrPtr(i)))) DO
            i := i + 1;
        {idem da destra}
        WHILE (j > lo) AND
             (comp(extr(pivot), extr(arrPtr(j)))) DO
          j := j - 1;
        IF (i < j) {se sono due elementi distinti,
                    scambiali tra loro, altrimenti
                    scambialo con il primo dell'array}
          THEN
            Swap (i, j)
          ELSE
            Swap (lo, j);
      END;
    FindPosOfPivot := j;
  END;{FindPosOfPivot}
```

ArrPtr viene utilizzata da **FindPosOfPivot** per ottenere il puntatore all'elemento di indice "ind" nell'array da ordinare

```
FUNCTION ArrPtr(ind: integer) : Ptr;

  BEGIN
    ArrPtr := POINTER (ORD4(arrayToSort) +
                       ORD4(ind)*ORD4(DimOfRec));
  END;
```

Swap scambia gli elementi di posto i e j nell'array:

```
PROCEDURE Swap(i, j : integer);

  BEGIN
    {Metti in temp l'i-esimo elemento}
    BlockMove(ArrPtr(i), temp, ORD4(DimOfRec));
    {Copia il j-esimo elemento nell'i-esimo}
    BlockMove(ArrPtr(j), ArrPtr(i), ORD4(DimOfRec));
    {Copia temp nel j-esimo elemento}
    BlockMove(temp, ArrPtr(j), ORD4(DimOfRec));
  END;
```

Nelle routines di Editing vengono utilizzate due procedure, **convfromscrap** e **convtoscrap**, che operano le conversioni opportune tra formato record e formato testo (quello dello scrap). **Convfromscrap** data una stringa di scrap, la converte in un record, interpretando come separatori di campi i TAB e i Return. Si noti che sono stati tralasciati i controlli, per altro necessari sulla completezza delle informazioni contenute nello scrap. Per ogni campo letto, infatti, occorrerebbe verificare la fine della stringa di scrap.

```
PROCEDURE convfromscrap(VAR astr: STR255;
                        VAR aPers: person);

  VAR
   scan     : INTEGER; {Indice nella stringa aStr }
   endofstr : Boolean; { Si è arrivati alla fine della
                         stringa }
  BEGIN
   scan := 1;

{ Inserisce nei campi di aPers le sottostringhe di aStr
  fra un delimitatore e l'altro fino a che non raggiunge
  la fine di aStr }
   WITH aPers DO
    BEGIN
     endofstr := gettoken(@nome, SIZEOF(nome) - 1);
     {Qui manca il controllo se endofstr ha valore True, e
      quindi lo scrap contiene un indirizzo incompleto (o
      dei dati incompatibili)}
     endofstr:= gettoken(@cognome, SIZEOF(cognome) - 1);
     endofstr := gettoken(@citta, SIZEOF(citta) - 1);
     endofstr := gettoken(@via, SIZEOF(via) - 1);
     endofstr := gettoken(@cap, SIZEOF(cap) - 1);
     endofstr := gettoken(@tel, SIZEOF(tel) - 1);
     deleted := False;
     GetDateTime(data);
    END;
  END;
```

Gettoken, dato un puntatore di destinazione ed una lunghezza massima, a partire dal valore attuale di scan inserisce nel campo di destinazione i caratteri di aStr (eliminando gli spazi iniziali) fino a che non incontra un delimitatore o supera la lunghezza MaxLen.

```
  FUNCTION gettoken(Dest: Stringptr;
```

```
                    MaxLen: INTEGER): Boolean;

VAR
 i   : INTEGER;{ pos. corrente nella stringa di dest. }

BEGIN
 i := 1;

 { Salta gli spazi iniziali }
 WHILE (astr[scan] = ' ') AND (scan <= Length(astr)) DO
  scan := scan + 1;

{ Copia finchè non trova un delimitatore (chr(9)=TAB,
  chr(13)=Return) o raggiunge la dimensione massima
  della stringa di destinazione }
 WHILE (scan <= Length(astr)) AND
       (astr[scan] <> CHR(9)) AND
       (astr[scan] <> CHR(13)) DO
  BEGIN
   { Finchè non si supera MaxLen copia i caratteri }
   IF i <= MaxLen
    THEN
     BEGIN
      Dest^[i] := astr[scan];
      i := i + 1;
     END;

   { Comunque prosegue fino al prossimo delimitatore }
   scan := scan + 1;
  END;

{ Aggiorna la lunghezza della stringa }
 Dest^[0] := CHR(i - 1);

{ La riempie di spazi fino alla fine }
 blankfill(Dest, MaxLen + 1);

{ Si è arrivati alla fine di aStr ? }
 gettoken := scan > Length(astr);

{ Passa al primo carattere dopo il delimitatore }
 scan := scan + 1;
END;
```

Ben più semplice è la routine inversa, **convtoscrap**, che carica nella stringa passata come parametro il record corrente. Si fa uso qui della funzione Pascal *Concat*, concatenando i campi del record e separandoli con un Return.

```
PROCEDURE convtoscrap(VAR astr: STR255);

  VAR
   acr      : string[1];

  BEGIN
   acr := ' ';
   acr[1] := CHR(CR);
   WITH thetabhdl^^[theindhdl^^[currecind]] DO
    astr := Concat(nome, acr, cognome, acr, citta, acr,
                   via, acr, cap, acr, tel, acr);
  END;
```

Nella sezione di docommand relativa alla ricerca, che abbiamo visto parlando del relativo dialog, si demanda tutto il lavoro alla funzione **dosearch**, che ritorna un valore negativo se la ricerca è fallita o positivo, uguale al numero d'ordine dell'elemento trovato, nel caso contrario. La ricerca viene effettuata secondo il tipo di ordinamento corrente ed è disabilitata nel caso in cui esso sia per data di immissione del record.

```
 FUNCTION dosearch: INTEGER;

  VAR
   i          : INTEGER;
   tobefound, continue: Boolean;
   astrptr  : ptr;
   anint    : INTEGER;

  BEGIN
   continue := TRUE;    {cambia in False se ritorni al
                       record da cui sei partito a cercare}
   tobefound := TRUE;   {deve ancora essere trovato?}
   i := currecind + 1;
   IF i >= numrec
{Se supero l'ultimo record, la ricerca riprende dal primo}
    THEN i := 0;

   WHILE tobefound AND continue DO
    BEGIN
     astrptr := access(@theindhdl^^[i]);
     {findstr cerca la posizione della stringa di ricerca
      nel campo puntato da astrptr; il valore ritornato è
      la posizione di inizio nella stringa o -1 se la
      stringa non compare}
     anint := findstr(Stringptr(astrptr), @searchstr);
```

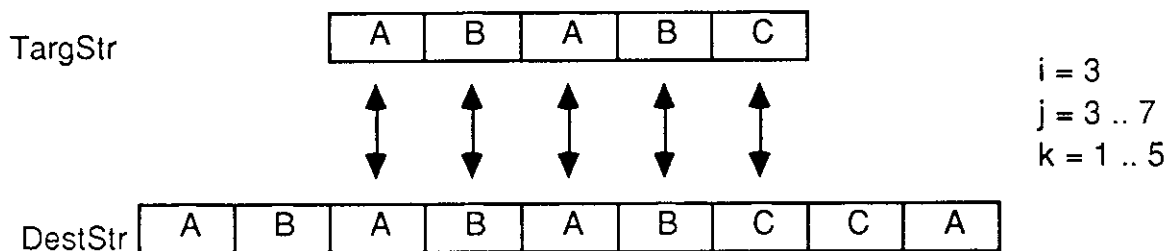
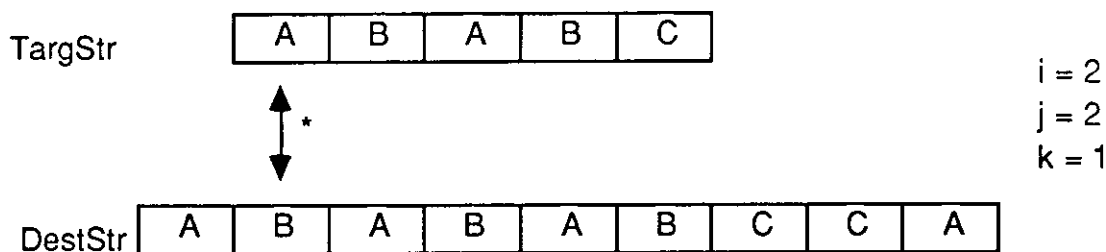
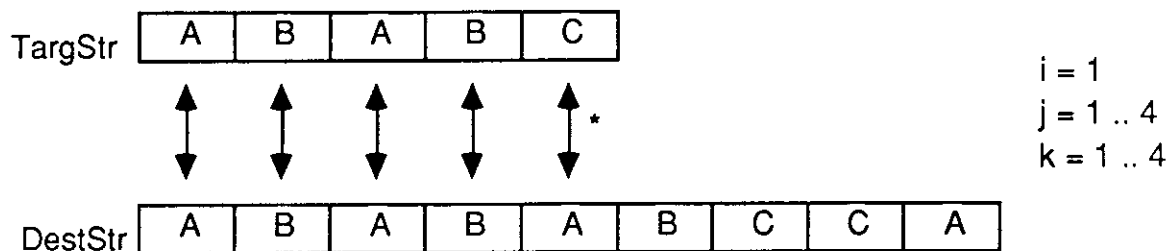
```
      { se è la stringa iniziale del campo del record
        oppure fa parte del campo e il tipo di ricerca è
        "CONTIENE", allora è ok.}
      IF (anint = 1)
         OR ((anint > 0) AND (NOT issearchinit))
       THEN tobefound := False  {non devi più cercare}
       ELSE
        BEGIN
         IF i = currecind
         { Siamo ritornati al punto di partenza }
          THEN continue := False;
         i := i + 1;
         IF i >= numrec
          THEN i := 0; { Si riprende dal primo record }
        END;
     END;

    {se, nonostante tutto, non hai trovato il campo
     richiesto, segnalalo al livello superiore}
    IF tobefound
     THEN dosearch := - 1
     ELSE dosearch := i;
   END;
```

Il confronto tra le due stringhe (DestStr del record e TargStr di ricerca) viene fatto da **findstr**, il cui valore è la posizione di partenza di TargStr in DestrStr, se è contenuta, o -1 altrimenti. Poichè Deststr è di lunghezza maggiore o uguale a TargStr, occorre confrontare per ogni posizione di differenza se le due stringhe combaciano.

Per esempio, se DestStr è di 10 caratteri e TargStr di 8, esse possono corrispondere a partire dal primo carattere di DestStr, dal secondo o dal terzo, ma non dal quarto, perchè TargStr non sarebbe più contenuta completamente.

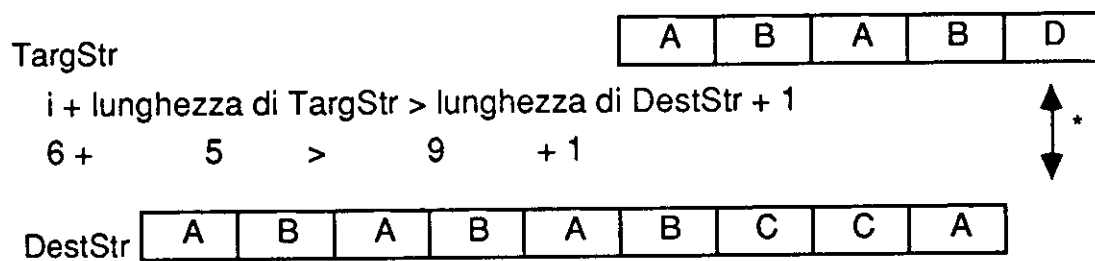
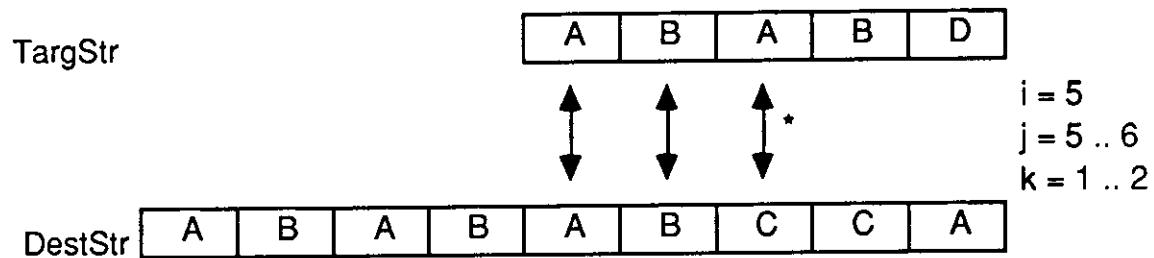
Allora a partire da ogni possibile posizione di inizio della corrispondenza (i), vengono confrontati i caratteri di DestStr (secondo l'indice j, che parte appunto da i) con i caratteri di TargStr, secondo l'indice k (che invece riparte ogni volta da 1). I due cicli del confronto terminano non appena le stringhe corrispondono, oppure quando le posizioni di inizio possibili sono esaurite (cioè i >= Length (DestStr) - Length (TargStr) + 1).

```
FUNCTION findstr(DestStrP, TargStrP: Stringptr): INTEGER;

  VAR
   i,             {Indice di dove si suppone che TargStr
                   inizi in DestStr}
   j,             {indice di scansione in DestStr}
   k  : INTEGER;  {indice di scansione in TargStr}
   found     : Boolean;

  BEGIN
   i := 0;
   found := False;
   WHILE (i < (Length(DestStrP^) - Length(TargStrP^) + 1))
          AND NOT found DO
    BEGIN
     i := i + 1;
     j := i;
     k := 1;
     WHILE (DestStrP^[j] = TargStrP^[k]) AND NOT found DO
      IF k = Length(TargStrP^)
       THEN found := TRUE
       ELSE
        BEGIN
         j := j + 1;
         k := k + 1;
        END;
    END;

   IF found
    THEN findstr := i
    ELSE findstr := - 1;
  END;
```

Nel seguito vediamo in una figura come viene realizzato il confronto fra la stringa "target" ABABC e la stringa di "destinazione" ABABABCCA. Nella prima fase il ciclo di confronto dei caratteri si ferma dopo 4 iterazioni, quando confronta C nel target con A nella stringa di destinazione; quindi l'indice i viene incrementato e subito si ferma il nuovo ciclo di confronto; infine con i = 3 il ciclo di confronto termina con successo, avendo avendo determinato che tutti i caratteri del target (k = 5) corrispondono ai caratteri della stringa di destinazione.

Fasi di confronto dei caratteri nella ricerca di una stringa

Una ricerca con esito negativo sarebbe terminata non appena la lunghezza del target più il valore di i fosse divenuta maggiore della lunghezza della stringa di destinazione più uno. In questo caso infatti l'ultimo carattere del target non potrebbe comunque essere confrontato con la stringa di destinazione.

TargStr
A
B
A
B
D
i = 5
j = 5 .. 6
k = 1 .. 2
DestStr
A
B
A
B
A
B
C
C
A

TargStr
A
B
A
B
D
i + lunghezza di TargStr > lunghezza di DestStr + 1
6 + 5 > 9 + 1
DestStr
A
B
A
B
A
B
C
C
A

5 I sistemi di sviluppo

In questo capitolo ci occuperemo dei sistemi di sviluppo che sono stati utilizzati per realizzare il programma di esempio. La scelta fra i molti sistemi disponibili oggi per lo sviluppo su Macintosh è caduta sui quattro che in questo momento sembrano essere i più interessanti:

- Il Pascal in ambiente Workshop su Lisa; è stato il primo sistema di sviluppo per Macintosh ed è tuttora il più completo e diffuso. Ha rispetto ad altri sistemi l'enorme vantaggio di essere il sistema con cui è stato pensato e realizzato tutto il software di base di Macintosh. Il problema maggiore invece è dovuto al fatto che Lisa non viene più prodotto nè venduto, ma è stato anche annunciato il prossimo rilascio di questo ambiente operante direttamente su Macintosh. Nel capitolo relativo al Toolbox si è sempre fatto riferimento al programma realizzato in Lisa Pascal.

- Il sistema TML Pascal, un compilatore Pascal quasi (nella versione attuale) completamente compatibile con il Lisa Pascal, che permette quindi di utilizzare già ora il Mac come sistema di sviluppo, riutilizzando tutto il software sviluppato su Lisa. Un'altra caratteristica che lo rende particolarmente interessante è il prezzo ($99).

- Il BASIC della MicroSoft, è stato inserito in questa rassegna più per la diffusione del linguaggio BASIC in sè e in particolare di questo prodotto che per le caratteristiche del linguaggio, peraltro abbastanza limitate. Il sistema è un interprete in grado di gestire con costrutti appositi alcune delle caratteristiche dell'interfaccia Macintosh come finestre, menù e operazioni grafiche.

- Il sistema Aztec-C, un sistema di sviluppo con interfaccia Unix-like basato sul linguaggio C. Il sistema è stato presceto per il particolare interesse del linguaggio C, che permette di sviluppare applicazioni compatte ed efficienti senza rinunciare alle strutture di controllo dei linguaggi ad alto livello. Fra tutti i sistemi di sviluppo basati sul linguaggio C abbiamo utilizzato il sistema Aztec perchè da una prima analisi è risultato il più interessante come prodotto e perchè attualmente è l'unico che viene supportato oltre che distribuito in Italia.

Workshop Pascal

Il workshop è nato come sistema di sviluppo su Lisa di applicazioni per Lisa; in seguito al sistema originario sono state aggiunte una serie di caratteristiche che hanno permesso di utilizzarlo come sistema di sviluppo di applicazioni per Macintosh.

Nel Workshop è compreso il sistema operativo di Lisa e il sistema permette di sviluppare ed eseguire applicazioni su Lisa oppure di sviluppare applicazioni che verranno poi eseguite su Macintosh. Fornisce gli strumenti necessari per la scrittura, il debugging e l'esecuzione di programmi Pascal oltre a una serie di strumenti per il supporto allo sviluppo.

L'interazione con il sistema avviene per mezzo di una struttura a menù di tipo simile a quella dell UCSD Pascal. Il sistema è provvisto di un ottimo editor multi-window con uso del mouse, di un sottosistema di file managing e inoltre permette l'uso di exec files, cioè di sequenze di comandi che vengono eseguiti in batch.

Lo sviluppo di una applicazione Macintosh su Lisa segue le seguenti fasi: scrittura o modifica del programma e del file di risorse con l'editor, compilazione con il compilatore Pascal o l'assemblatore, collegamento dei vari moduli alle librerie con il linker, compilazione del file di risorse e inclusione del codice generato dal linker con il programma RMaker, che scrive il risultato su un dischetto 3" 1/2 in formato Macintosh. Il programma così ottenuto può essere eseguito su un Macintosh o su un Lisa in emulazione Mac con MacWorks.

Nel sistema di sviluppo del Workshop è compresa una serie di files che permettono l'interfacciamento con il Toolbox di Macintosh. Questi sono caratterizzati dal prefisso del nome del file:

- **Intrfc/** Files di testo contenenti l'interfaccia Pascal alle routine e ai tipi di dati utilizzati dal Toolbox di Macintosh.

- **TlAsm/** Files di testo da includere quando si usa l'assembler

che contengono le macro e le definizioni delle costanti di sistema di Macintosh.

- **Obj/** Files oggetto. Sono units precompilate in cui viene definita la parte di codice che interfaccia il Pascal al Toolbox.

- **Example/** Files di testo contenenti esempi di programmi che vengono forniti dal Macintosh Technical Support.

Il dialetto Pascal accettato dal compilatore è ragionevolmente standard e comprende alcune interessanti estensioni come:

- interi a 32 bit
- clausola **otherwise** negli statement **CASE**
- passaggio di parametri procedurali e funzionali
- un operatore unario @ che restituisce il puntatore a un oggetto
- la conversione di tipi negli assegnamenti (purché delle stesse dimensioni).
- un costrutto **inline** che per la definizione diretta di codice in formato esadecimale
- la possibilità di definire puntatori di tipo **univ** come parametri di una procedura o una funzione. Una funzione con un parametro definito come **univ ptr** potrà accettare come parametro effettivo un puntatore di tipo qualsiasi.

Inoltre il compilatore fornisce l'uso di direttive di compilazione per mezzo di pseudo-commenti (commenti che iniziano con il carattere $ immediatamente seguito dal carattere che specifica la direttiva) inseriti nel codice. Con questo strumento é possibile controllare alcune opzioni come l'inserimento di codice per il range checking delle variabili e degli indici di array, controllo degli overflow a run-time, inclusione di altri files, definizione di segmenti separati di codice e costrutti per la compilazione condizionale di parti di codice.

Il compilatore del Lisa Pascal permette anche la definizione e la compilazione separata di **Units**. Una unit è un file oggetto non

eseguibile che può essere collegato ad altri file oggetto per produrre programmi completi. Ogni unit usata da un programma (o da un'altra unit) deve essere compilata e il codice oggetto deve essere accessibile al compilatore prima di poter compilare il programma principale (o la unit) che lo usa. Le unit vengono usate per modularizzare programmi di grandi dimensioni e permettono ad esempio, quando si effettua una modifica, di non ricompilare l'intero programma ma solo la unit modificata e quelle che la usano.

Quando un programma o unit (detto host) usa un'altra unit, il linker inserisce una copia del codice compilato della unit nel codice oggetto dell'host. Una unit si compone di tre parti: una *intestazione*, una parte di *interfaccia* e una parte di *implementazione*. La parte di interfaccia definisce costanti, variabili, procedure e funzioni che sono pubbliche, cioé visibili dall'host. L'host può accedere a queste entità come se fossero state dichiarate all'interno del suo codice. La parte di implementazione contiene la dichiarazione di tutte le entità che devono restare "private" alla unit e ladefinizione effettiva delle procedure e delle funzioni dichiarate nella parte di interfaccia della unit.

Nel Workshop si possono utilizzare le routine e le strutture dati del Toolbox per mezzo delle units. Infatti i files che vengono distribuiti con il prefisso **obj/** sono le units relative alle varie sezioni del Toolbox di Macintosh. Ogni programma che utilizza il Toolbox dovrà "usare" queste units, quindi nell'intestazione dei programmi scritti con il Lisa Pascal troveremo sempre una lista delle units che vengono utilizzate. Ad esempio il nostro programma di esempio utilizza alcune delle units del toolbox e la unit relativa alle routine di ordinamento da noi definita (unit qsort nel file Apple-Qsort).

```
USES {$U-}
  {$U Obj-Memtypes     } memtypes,
  {$U Obj-QuickDraw    } quickdraw,
  {$U Obj-OSIntf       } osintf,
  {$U Obj-ToolIntf     } toolintf,
  {$U Obj-PasLibIntf   } paslibintf,
  {$U Obj-packIntf     } packintf,
  {$U Obj-MacPrint     } MacPrint,
  {$U Apple-Qsort      } qsort;
```

Il Workshop mette a disposizione dello sviluppatore alcuni strumenti per rendere più agevole la stesura e la correzione del codice e il trasferimento dell'applicazione su Macintosh.

Durante lo sviluppo di una applicazione per Mac si utilizza un compilatore di risorse chiamato RMaker come l'equivalente che gira su Mac. A questo programma va fornito il file delle risorse, che contiene, oltre alle risorse che si intendono definire, il nome del file oggetto generato dal linker da cui va prelevato il codice dell'applicazione e il nome del file da generare. Il file generato da RMaker verrà poi trasferito su un dischetto con il programma MacCom che permette di vedere e trasferire files da (a) formato Lisa a (da) formato Macintosh.

Tra gli strumenti più potenti e interessanti sono gli *exec files.* Attraverso il loro utilizzo si possono rieseguire sequenze complesse di comandi con la semplice invocazione del nome del file di exec; inoltre é possibile sfruttare, in questi files, strutture di controllo del tipo IF THEN ELSE, passare parametri, utilizzare variabili locali, eseguire dell I/O, richiamare altri exec e fare test sull'esistenza o sulla data di modifica di files. Con l'uso degli exec file il programmatore è quindi in grado di creare dei programmi per il *controllo* del sistema. Tra i file di esempio forniti con il Workshop c'é il file di exec example/exec che controlla tutte le operazioni (compilazione, generazione del codice, assemblaggio, linking, compilazione delle risorse e trasferimento a dischetto Macintosh) necessarie alla creazione di una applicazione standard per Macintosh; questo exec può essere convenientemente modificato e utilizzato per gestire lo sviluppo di applicazioni più complesse.

Altri programmi di utilità forniti con il sistema di sviluppo sono: Pasmat un formattatore di programmi Pascal con la possibilità di scegliere fra molte opzioni utili a personalizzare lo stile di formattazione, una serie di programmi per il cross reference di programmi Pascal (dove vengono definiti e dove vengono usati i simboli di un programma), un disassemblatore e un programma di comunicazione

TML PASCAL: Il linguaggio

Dopo aver visto lo standard Pascal primitivo in ambiente Lisa, vediamo le differenze di implementazione del TML, che lo distinguono dal Lisa Pascal. Tali differenze emergono anche dal confronto tra le due versioni del programma esempio.

Una prima grande differenza è generata dal fatto che non è permesso il passaggio di parametri procedurali e funzionali alle procedure dichiarate in Pascal. Ciò è dovuto a una non meglio identificata "Implementation Restriction"; in pratica si rimanda alla prossima versione del compilatore. Rimane valida comunque la possibilità di passare i ProcPtr alle routine del Toolbox, utilizzando l'operatore unario @. Sempre per ciò che riguarda le funzioni, occorre notare che il risultato di una funzione non può essere immediatamente utilizzato come puntatore per accedere ad un oggetto, ma va prima opportunamente assegnato ad una variabile.
Alcuni grossi problemi vengono dall'uso della funzione standard *sizeof*. Infatti non è applicabile a identificatori di tipo: per conoscere le dimensioni di un record (per esempio) occorre dichiarare (e quindi far allocare) una variabile di quel record. Questa è stata la procedura seguita nella "traduzione" del programma in TML, ma ne è nata una spiacevole sorpresa: il tipo Persontype è di dimensione logica dispari (155 bytes), perchè il Boolean deleted viene calcolato della dimensione di un byte, ma fisicamente occupa 156 bytes, perchè per il 68000 gli indirizzi agli oggetti in memoria devono sempre essere pari. Quando la tabella riferita da TheTabHdl viene allargata (tramite la routine *SetHandleSize*), la *sizeof*(Person), utilizzata per calcolare la nuova dimensione, restituisce 155. In tal modo viene allocata meno memoria del necessario e, dopo alcuni ridimensionamenti, i record "sbordano" al di là delle dimensioni fisiche del blocco allocato per la tabella, e questo genera, alla fine, una "bomba" per "address error". La modifica apportata consiste nel rendere pari anche la dimensione logica del record, aggiungendo un filler esplicito, che costringe la *sizeof* a fare i conti correttamente.

Un altro grosso problema è sorto nel riconoscimento della stringa nulla. Nella versione TML il confronto con il literal '' è stato sostituito con il test sul byte 0 della stringa (il campo lunghezza)

Un'altra limitazione nell'implementazione del Pascal viene dalla assenza dell'istruzione di **exit**, che, anche se non elegantissima, permette di sospendere una routine al momento opportuno senza test aggiuntivi inutili. Essa può essere sostituita con una **goto** (ma Wirth non sarebbe molto d'accordo) oppure invertendo il test (come nel caso del programma di esempio) e racchiudendo le cose da non fare tra **begin** ... **end**. Cioè:

```
Lisa                                  TML
BEGIN                           BEGIN
   IF <condizione>                    IF  NOT <condizione>
         THEN EXIT;                         THEN
                                                  BEGIN
 ............                                     ................
 ............                                     ................
                                                  END;
END;                            END;
```

Una differenza minore, ma che può creare problemi in compilazione è la limitazione della cardinalità dei tipi SET ad un massimo di 32 elementi (addio SET OF char!!!). Occorre perciò sostituire i test su un SET unico con più test su SET di dimensioni inferiori.

TML Pascal: il sistema

La più grande differenza nel metodo di scrittura di un programma TML risiede nell'accesso alle variabili e alle routines del Toolbox. Esse vanno infatti dichiarate includendo nel programma i files che le contengono, e non linkando delle librerie esterne. Ciò significa sostituire le dichiarazioni di USES con delle direttive di Include in testa al programma. Ciò risulta necessario perché il TML non consente le compilazioni separate di UNIT, e quindi tutto il programma (dichiarazioni di toolbox comprese) va ricompilato totalmente ogni volta. Un altra mancanza abbastanza avvertibile viene dalla mancanza dei file di exec, che costringe l'utente ad attendere il completamento dei singoli passi (Fine editing, Compilazione, Linking, Compilazione delle risorse).

Ciò detto va anche osservato che questo è al momento il miglior compilatore Pascal funzionante su MacIntosh, e sicuramente ad un prezzo assai competitivo. Permette infatti la creazione di applicazioni stand-alone e di desk accessories e il codice prodotto si colloca come dimensioni e come tempi di esecuzione nella media degli altri linguaggi compilati.

MSBASIC - Il linguaggio.

Il nocciolo del programma per la gestione dell'archivio di indirizzi, sia nella versione Pascal che nella versione C, è formato da un unico ciclo durante il quale vengono eseguite le procedure legate agli eventi che vengono "trappati" dal sistema e segnalati poi al programma stesso. La gestione in MS-BASIC cambia di poco per quel che riguarda la "filosofia", ma in modo più marcato per ciò che riguarda le operazioni da svolgere. In pratica il programma in BASIC è costituito da un "fannullone" che viene risvegliato a tratti dall'insorgere di situazioni "strane" e dalle varie routines che si occupano di queste situazioni. Ovvero: il nocciolo del programma consiste solo in un ciclo (che in prima istanza si può considerare vuoto) che terminerà nel momento in cui una variabile di controllo (done) assumerà il valore TRUE; nel frattempo, ad interrompere l'"ozio" interviene in modo asincrono il sistema che segnala l'esistenza di eventi.

La parte principale (e più difficile da scrivere) del programma è l'insieme delle subroutines associate ad ogni tipo di evento che il sistema segnala tramite l'interprete: *BREAK, ERROR, MOUSE, MENU e DIALOG.*
L'evento di tipo *BREAK* è generato dalla combinazione di tasti shift-command-punto, ed ha come risultato generico quello di interrompere il programma; trappare questo evento, mascherandolo quindi all'inteprete, equivale a rendere non interrompibile il programma stesso dall'esterno (con gravi pericoli se non tutto funziona correttamente).
ERROR non è un evento vero e proprio, ma riprende il tradizionale metodo BASIC di segnalazione degli errori che si verificano durante l'esecuzione del programma (errori di sintassi, fallimenti nell'apertura o lettura o scrittura di files, indirizzamento di elementi di array oltre le dimensioni fissate, ecc.)
Un evento di tipo *MOUSE* viene segnalato quando vi sia stato un click semplice, doppio o triplo in una finestra attiva. Per mezzo della funzione *MOUSE()*, con diversi parametri, si arriva a stabilire dove i click siano stati effettuati.
Un evento di tipo *MENU* viene segnalato nel momento in cui viene selezionato con il mouse un item abilitato di un menù. *MENU(0)* e

MENU(1) provvedono poi ad informare su quale menù e quale item siano stati selezionati.
Gli eventi di tipo *DIALOG* sono quelli legati alla comunicazione con le finestre, non solo quelle di dialogo vere e proprie, ma anche le finestre di tipo documento.

Il manuale che correda l'interprete contiene alcune pagine che illustrano il funzionamento del meccanismo di event trapping, osservando anche quali siano gli inconvenienti cui è possibile andare incontro nell'utilizzare le subroutine di gestione degli eventi (dette comunemente handlers). Tali errori sono purtroppo determinati in larga parte dalla totale condivisione delle variabili all'interno dei programmi BASIC. Poichè le chiamate agli handlers sono assolutamente asincrone, nulla di più facile che, per disattenzione, si utilizzi la stessa variabile all'interno di due handlers diversi, modificandone il valore in uno e lasciandola inconsistente per gli usi dell'altro.
Esiste però un secondo metodo, certamente più sicuro, ma anche in un certo senso inelegante e causa di difficile leggibilità del programma, che per altro è consigliato dalla stessa MS: il polling. Si tratta di trasformare il programma in un ciclo durante il quale si continua a testare la presenza sulla coda di eventi di un certo tipo, prendendosi carico della gestione di essi uno alla volta.
Alcune complicazioni nascono al secondo livello di dialogo, quando sulla finestra "base" occorre aprirne una seconda per la richiesta dei nuovi record o della stringa di ricerca. Occorre allora definire un secondo handler che gestisca la finestra di tipo modal, che contiene bottoni e campi di editing (EditText Box). Tale handler deve farsi carico delle seguenti situazioni: click in un bottone, click per selezionare un campo di editing, pressione di Return o Enter (che non viene interpretata automaticamente dal BASIC come un click nel bottone di default), pressione di un TAB (che non corrisponde automaticamente alla selezione del campo successivo di editing). Se poi, come è necessario in un paio di occasioni, bisogna aprire una terza finestra (di alert)...

Nella versione BASIC del programma il metodo di ordinamento (il quicksort) è stato sostituito dallo shellsort, visto che la ricorsività in BASIC viene lasciata tutta a carico dell'utente e comporta un aggravio di complessità del programma, causata dalla necessità

di gestire esplicitamente lo stack su cui vengono memorizzati i parametri.

In generale l'interfaccia diretta con il Toolbox risulta molto scarsa, essendo state sostituite le principali chiamate a sistema con funzioni proprie dell'interprete. Dove esse mancano, le carenze affiorano a prima vista, infatti, per esempio, mancano le primitive per l'utilizzo delle barre di controllo, delle icone, e dei keyboard equivalent (che nell'esempio sono stati simulati per mezzo di un polling sulla tastiera)

La gestione della memoria appare piuttosto ridotta all'essenziale: non viene data la possibilità di utilizzare dirtettamente gli handles (ma comunque le strutture dati vengono allocate in blocchi rilocabili) e gli array non possono essere variati nelle dimensioni, ma solo (eventualmente) cancellati e ridimensionati; er ogni dimensione il numero massimo di elementi dipende dalle dimensioni del tipo base (interi due bytes, singola precisione quattro bytes, ecc.), ma comunque fino ad un massimo di 64K. Il numero di dimensioni invece può variare fino ad un massimo di 255.

Alcuni problemi nascono dal meccanismo di output. Infatti le istruzioni di *PRINT* si riferiscono sempre alla finestra corrente di output, che per default è la finestra attiva. Se occorre qualche operazione di output su una finestra non attiva (come nel caso del refresh della tabella dei record, nell'esempio), occorre dichiararla temporaneamente di output, badando poi, al termine dell'operazione, di ridichiarare la finestra attiva come anche finestra di output. Probabilmente una estensione alle primitive di output con l'introduzione della parametrizzazione rispetto alla finestra (come nel Toolbox) avrebbe evitato queste complicazioni.

Una novità rispetto al BASIC standard viene dalla possibilità per il programmatore di definire dei sottoprogrammi, richiamabili attraverso chiamate (*CALL*) con passaggio di parametri. I sottoprogrammi possono essere di tipo predefinito (e sono alcune chiamate al Toolbox), oppure definiti dall'utente in BASIC, oppure infine possono essere routines in linguaggio macchina, sempre fornite dall'utente.

I sottoprogrammi del primo tipo (predefiniti) sono circa una cinquantina, e permettono l'utilizzo delle principali routines di Quickdraw, per la gestione delle figure, del cursore, della penna grafica e delle caratteristiche del testo.
Per ciò che riguarda il secondo tipo di sottoprogrammi, il linguaggio ammette una sintassi del tipo:

SUB <nome> (parametri) *STATIC*
...
...
END SUB

che consente di racchiudere parti di programma indipendenti; infatti i sottoprogrammi così definiti "vedono" dell'ambiente esterno solo le variabili passate come parametro, e quelle esplicitamente definite *SHARED* all'interno del sottoprogramma stesso. Il passaggio dei parametri avviene per valore se si tratta di espressioni, o per riferimento (come *var* in Pascal) se si tratta di variabili singole. Il passaggio di una variabile singola per valore viene indicato all'interprete racchiudendo la variabile tra parentesi e simulando quindi una espressione.
I sottoprogrammi del terzo tipo sono infine gli unici che permettono di accedere completamente al Toolbox, in quanto permettono di definire e richiamare delle routines in linguaggio macchina. Tali routines vanno caricate in strutture dati BASIC (tipicamente array di interi) e infine lanciate per mezzo di *CALL*. Le routines possono essere assemblate "a mano" oppure lette da files ottenuti dall'assemblatore ASM del Macintosh Development System.

Un'altra novità interessante rispetto ai BASIC tradizionali è la non obbligatorietà di numerare le linee di istruzioni e la possibilità di utilizzare anche delle labels alfanumeriche. Ciò permette di associare degli mnemonici significativi alle subroutines, incrementando la leggibilità del programma.

Per ciò che attiene la gestione dei files l'MS-BASIC, oltre alle tradizionali primitive come *OPEN, INPUT#, PRINT#, FILES*, mette a disposizione anche una funzione per richiamare i due dialog dello Standard File Package, *FILES$*. In particolare,

FILES$(0,MSG$) permette di richiedere all'utente il nome di un file, che per esempio andrà memorizzato su disco, presentandogli il dialog con il messaggio MSG$;
FILES$(1,TYPE$) permette invece di far selezionare un file tra quelli presenti su disco che siano di uno dei tipi specificati nella stringa TYPE$ (nell'esempio solo i files di tipo "ARCH").
Tutti i files creati da un programma BASIC sono inizialmente di tipo 'TEXT'. Per modificarlo, è possibile utilizzare l'istruzione

```
NAME OLDNAME$ AS NEWNAME$,NEWTYPE$
```

che permette di rinominare il file in oggetto, riattribuendogli anche il tipo.

Un file particolare è rappresentato dalla clipboard, che può essere utilizzata operando sul file "CLIP:" (si veda a tale proposito la gestione di **MYCOPY** e **MYPASTE** nell'esempio).

Un'ultima osservazione va fatta per ciò che riguarda la stampa. La tradizionale sintassi, che prevede l'apertura del file LPT1: per la stampante, è stata estesa, prevedendo l'apertura di "LPT1:PROMPT", per l'utilizzo della dialog box di aggiornamento delle caratteristiche della stampa.

MS-BASIC - Il sistema di sviluppo.

Il sistema di sviluppo MS-BASIC non consente la creazione di applicazioni stand-alone, non essendo dotato di un compilatore, e cerca di sopperire a questa lacuna ampliando al massimo i mezzi a disposizione del programmatore per la scrittura e il debugging dei programmi.
Permette infatti l'apertura di due finestre su cui ottenere il LISTing del programma; non è possibile purtroppo utilizzare questa opportunità per operare su più files, essendo ammessa la presenza di uno solo per volta. Un'altra finestra può venire aperta contemporaneamente alle precedenti, ed è quella di command, che permette l'esecuzione in modo immediato delle istruzioni BASIC. Essa risulta molto utile quando un programma utente che ha modificato la barra dei menù è stato interrotto con un command-shift-period: infatti l'unico modo di ripristinare la situazione è far eseguire (dalla finestra di command) l'istruzione *MENU RESET*.
Infine l'interprete apre automaticamente all'inizio di ogni programma una finestra di output, cui viene associato l'ID 1.

Il sistema permette l'interazione tra programma e programmatore per mezzo della funzione di TRACE, richiamabile sia all'interno del programma (*TRON/TROFF*), sia per mezzo di un item del menù RUN. Spostando poi la finestra di output del programma su un lato dello schermo (cosa che non crea problemi, in quanto il dragging è completamente gestito dall'interprete) è possibile seguire sulla finestra di LIST (opportunamente ampliata) le singole istruzioni eseguite dall'interprete.
Un (brutto) neo del sistema è rappresentato dall'editor residente sulla finestra di LIST, che dovendo controllare sintatticamente ogni linea di codice introdotta, risulta molto lento, soprattutto nelle operazioni di Cut/Copy/Paste. Occorre segnalare anche che non sempre le selezioni estese (di più parole o linee) hanno l'effetto desiderato, proprio per la lentezza di cui si diceva; è consigliabile perciò verificare sempre la situazione prima di procedere ad operazioni di editing distruttive.
Un altro difetto è l'impossibilità di accedere contemporaneamente ad altri programmi (anche solo in read-only), rendendo così complesso il trasporto di subroutines già codificate e testate. E'

chiara in ogni modo la propensione del sistema a favorire i "vecchi" programmatori, ancora adusi a tecniche pre-mouse, per i quali val più la possibilità di utilizzare una buona maschera di stampa (*PRINT USING...*) rispetto ad una ben più elegante (e concisa) finestra di dialogo. Bisogna comunque sottolineare lo sforzo di MS di adeguare un linguaggio così scarsamente strutturato come il BASIC ad un ambiente come MAC, in cui invece le strutture si sprecano, senza tuttavia tentare sterili forzature del linguaggio originale, ma fornendo solo nuove primitive per l'accesso agli oggetti del Toolbox (anche se non a tutti, e in modo sempre parziale, come si è visto). Tale sforzo è assai importante, ma non deve essere altro che uno stimolo per chi, esperto del linguaggio BASIC, vuole capire prima le potenzialità della macchina, per poi passare, con lo studio di un linguaggio di livello superiore (Pascal o C) ad utilizzare a fondo (e con il massimo dei risultati) tutte le routines che il Toolbox mette a disposizione.

MS-BASIC - L'esempio.

Preferendo l'eleganza e la leggibilità anche a rischio di confusione, abbiamo scelto di implementare la versione BASIC del programma di esempio utilizzando gli handler per la gestione degli eventi: in particolare, MOUSEDPTCH per i click, DIALOGDPTCH per la finestra su cui vengono trascritti i record e DOCOMMAND per i MENU.

Il programma inizia disabilitando tutti gli eventi e chiudendo la finestra di default

```
MENU OFF:MOUSE OFF:DIALOG OFF
DEFINT A-Z
WINDOW CLOSE 1
TRUE=1=1
FALSE=1=0
GOSUB INIT
```

Dopo l'inizializzazione delle variabili, dei menù e della finestra vengono riabilitati gli eventi e il programma si mette in attesa di eventi o di caratteri (di controllo) da keyboard.

```
ON MENU GOSUB DOCOMMAND:MENU ON
ON MOUSE GOSUB MOUSEDPTCH: MOUSE ON
ON DIALOG GOSUB DIALOGDPTCH: DIALOG ON
10
 A$=INKEY$
 IF A$="" THEN 20
 IF A$<" " THEN GOSUB KEYBEQ ELSE BEEP
20  IF NOT DONE THEN 10
GOSUB CLOSEALL
END
```

Le combinazioni di caratteri con Command vengono tradotte dall'interprete come caratteri di controllo del set ASCII. In questo modo sono riconoscibili i keyboard equivalents

```
KEYBEQ:
 ON ASC(a$) GOSUB A, A, MYCOPY, DELACTRECORD, DOQUIT, A,
                  A, A, A, A, A, A, MODIFYRECORD, DONEW,
                  DOOPEN, A, A, NEWRECORD, A, A, A,
                  MYPASTE, A, MYCUT, A, UNDO
RETURN
```

```
A: BEEP: RETURN

CLOSEALL:
RETURN

INIT:
FILEID=1
EDITID=2
SORTID=3
SEARCHID=4
CMD$=CHR$(17)
```

Crea un menù (0)con ID fileid, attivo (1), di nome Archivio

```
MENU FILEID,0,1,"Archivio"
```

Crea un elemento (1) del menù FILEID, attivo (1), di nome "Nuovo"

```
MENU FILEID,1,1,"Nuovo                    "+CMD$+"N"
...
```

eccetera

```
...

MENU EDITID,0,1,"Edit"
MENU EDITID,1,0,"Undo                     "+CMD$+"Z"

...
...

MENU SORTID,0,0,"Ordinamento"
CURSORT = 1
MENU SORTID,1,2,"Per nome"
...
...

MENU SEARCHID,0,0,"Ricerca"
MENU SEARCHID,1,1,"Cerca"

MENU 5,0,0," "
REM NASCONDE IL MENU WINDOWS DELL'INTERPRETE

DELETED=0: REM "*" / " "
CAMPODATA=7: REM DATE$+" "+TIME$
REM DA 1 A 6 CI SONO I CAMPI NOME, COGNOME, ECC.
```

Definisci la dimensione della tabella

```
TABDIM=50
DIM THETAB$(TABDIM,7),THEIND(TABDIM)
FSTR$="\"+SPACE$(19)+"\": REM format string
NUMREC=0:CURRECIND=0
textheight=16
pagerec=16
GOSUB WINCREATE

RETURN

WINCREATE:
WINDOW 1, "Senza nome"
       (20,50) - (500,textheight * pagerec + 50), 1
DIRTY=FALSE
SCROLLCOUNT=0
THEFILE$=""
SEARCH$=""
RETURN
```

MOUSEDPTCH deve occuparsi dei click di selezione dei record. Un click semplice (MOUSE(0)=1 o -1) seleziona un record, mentre un doppio click (2 o -2) lo apre per aggiornarlo.

```
MOUSEDPTCH:
X= MOUSE(0)
IF X=1 OR X=-1 THEN GOSUB NEWRECIND:RETURN
IF X=2 OR X=-2 THEN GOSUB NEWRECIND:GOSUB MODIFYRECORD
RETURN
```

Modifyrecord verrà definita parlando di Docommand, mentre Newrecind, che segue, è identica nel comportamento all'omologa Pascal. Trovata con Mouse(4) la coordinata y del punto di click, l'indice del record selezionato viene trovato dividendo tale coordinata per l'altezza delle righe di testo e aggiungendo il valore di scrollcount, che tiene traccia degli eventuali scroll verso l'alto del testo. Nel caso in cui il record selezionato non sia quello corrente, la routine deseleziona la linea corrente e seleziona quella scelta.

```
NEWRECIND:
 IND=MOUSE(4)\ TEXTHEIGHT + SCROLLCOUNT
 IF IND=CURRECIND THEN RETURN
 GOSUB UNSELECTLINE
 CURRECIND=IND
 IF CURRECIND>=NUMREC THEN CURRECIND=NUMREC-1
 GOSUB SELECTLINE
```

```
 RETURN
```

Selectline modifica lo stile dei caratteri in sottolineato e richiama la routine unselectline, che scrive alla posizione corretta la linea il cui indice è contenuto in currecind.

```
SELECTLINE:
 CALL TEXTFACE(4)
 GOSUB UNSELECTLINE
 CALL TEXTFACE(0)
 RETURN

UNSELECTLINE:
 LOCATE CURRECIND-SCROLLCOUNT+1,1
 PRINT THETAB$(THEIND(CURRECIND),0);" ";
 IF CURSORT<=4 THEN LR=CURSORT ELSE LR=4
 FOR J=LR TO LR+3
   LOCATE CURRECIND-SCROLLCOUNT+1,(J-LR)*21+2
   PRINT USING FSTR$;THETAB$(THEIND(CURRECIND),J);
 NEXT
 RETURN
```

DIALOGDPTCH è ancora più semplice (a questo livello), perchè sulla finestra principale gli eventi di cui occorre occuparsi sono quelli di update e i click nella close-box (che per l'MS-BASIC è un evento di dialog), in quanto essa manca di bottoni, zone di editing e barre di controllo (queste ultime non si possono gestire per mezzo dell'interprete BASIC). Inoltre il dragging e il meccanismo di crescita/decrescita sono implicitamente a carico dell'interprete. La funzione Dialog(0) ritorna il tipo di evento accaduto sulla finestra, 4 se di update e 5 se di close. Nel primo caso occorre ridisegnare il contenuto della finestra (REFRESH), nel secondo comportarsi come nel caso della selezione di **Chiudi** nel menù **Archivio**.

```
DIALOGDPTCH:
 d=DIALOG(0)
 IF d=5 AND DIALOG(5)=1 THEN REFRESH
 IF d=4 AND DIALOG(4)=1 THEN DOCLOSE
 RETURN

REFRESH:
 sw=WINDOW(0)
 WINDOW 1
 GOSUB printrecs
```

```
WINDOW sw
RETURN
```

La routine PRINTRECS, che segue, si occupa di scrivere gli indirizzi sulla finestra a partire dal valore corrente di scrollcount e in modo tale da far comparire nella finestra il campo corrente di ordinamento.

```
PRINTRECS:
 IF NUMREC<=0 THEN RETURN
 Parti dal valore corrente di scroll
 FROMREC=SCROLLCOUNT
 Fino in fondo al file
  TOREC= NUMREC-1
 Se i record diventano troppi, riduci al numero massimo che può
  essere contenuto
 IF NUMREC-SCROLLCOUNT>PAGEREC
                                THEN TOREC = PAGEREC-1+SCROLLCOUNT
 Se il campo di ordinamento corrente è minore di 4, parti da
  quello a scrivere gli indirizzi, altrimenti parti comunque dal
  quarto.
 IF CURSORT<=4 THEN LR=CURSORT ELSE LR=4
 FOR I=FROMREC TO TOREC
 Se la linea che stai stampando è quella corrente, selezionala
  IF I=CURRECIND THEN TEXTFACE(4) ELSE TEXTFACE(0)
  LOCATE I-FROMREC+1,1
 Scrivi comunque il contenuto del campo deleted
     PRINT THETAB$(THEIND(I),DELETED); " ";
 e gli altri campi, a partire da quello prima calcolato
   FOR J=LR TO LR+3
     LOCATE I-FROMREC+1,(J-LR)*21+2
     PRINT USING FSTR$;THETAB$(THEIND(I),J);
  NEXT J
 NEXT I
 RETURN
```

DOCOMMAND ricalca strettamente la procedura omonima in Pascal, con l'analisi caso per caso dei menù e degli item (con le classiche istruzioni *ON...GOSUB*).

```
DOCOMMAND:
 Disabilita gli eventi
 MENU OFF:DIALOG OFF:MOUSE OFF
```

Da menu(0) e menu(1) si ottengono menù e item selezionati

```
  THEMENU= MENU(0)
 THEITEM= MENU(1)
```

A seconda del menù, esegui la routine relativa

```
 ON THEMENU GOSUB DOFILE,DOEDIT,DOSORT,DOSEARCH
```

quindi riporta in stato normale il menù selezionato

```
  MENU
```

Riabilita gli eventi

```
  MENU ON:DIALOG ON:MOUSE ON
RETURN
```

La routine per la gestione del menù **Archivio** è Dofile, che esegue la routine associata all'elemento selezionato

```
DOFILE:
ON THEITEM GOSUB DONEW, DOOPEN, DOCLOSE, DOSAVE, DOSAVEAS,
                 DOPSETUP, DOPRINT, DOQUIT
RETURN
```

La selezione di **Esci** attiva Doquit, che, assegnato valore True al flag Done, richiama Doclose, per la chiusura della finestra e l'eventuale salvataggio del file.

```
 DOQUIT:
  DONE= TRUE
  GOTO DOCLOSE
```

A sua volta Doclose controlla se la finestra è stata modificata dalla sua apertura (cioè Dirty vale True), nel qual caso viene attivata la routine Dosave. Comunque vengono azzerati i record presenti, riinizializzata la tabella degli indici e disabilitati gli opportuni item dei menù.

```
DOCLOSE:
  IF DIRTY THEN GOSUB DOSAVE: WINDOW CLOSE 1
  NUMREC=0
  FOR J=0 TO TABDIM
   THEIND(J)=J
  NEXT J
  MENU FILEID,3,0
  MENU FILEID,4,0
  MENU FILEID,5,0
  MENU FILEID,7,0
  MENU EDITID,0,0
  MENU SORTID,0,0
```

```
  MENU SEARCHID,0,0
  RETURN
```

Dosave controlla se esiste già un riferimento ad un file (contenuto nella variabile THEFILE$), nel qual caso trasferisce sul disco i record della tabella, altrimenti richiama la routine che gestisce anche **Salva come...**

```
DOSAVE:
   IF THEFILE$="" THEN DOSAVEAS ELSE SHOOTONTODISK
  RETURN
```

Dosaveas emette il dialog di *SFPutFile*, utilizzando la routine *File$*(0,<message>), e "cade" nella routine SHOOTONTODISK.

```
DOSAVEAS:
  Z$=FILES$(0,"SALVA CON CHE NOME?")
  IF Z$="" THEN RETURN
  THEFILE$=Z$
```

SHOOTONTODISK ha il compito di scrivere sul file gli indirizzi, evitando quelli marcati come cancellati (in cui cioè il campo deleted contiene un asterisco).

```
SHOOTONTODISK:
 Apri in output il file thefile$ con il numero 1
     OPEN "O",1,THEFILE$
 Per ogni elemento della tabella
     FOR I=0 TO NUMREC-1
  Se è cancellato, evita di scriverlo
         IF THETAB$(I,DELETED)="*" THEN SHNEXT
  Altrimenti scrivi i sei campi dell'indirizzo
       FOR J=1 TO 6
     PRINT#1,USING FSTR$;THETAB$(I,J)
    NEXT J
  e la data
       PRINT#1,THETAB$(I,CAMPODATA)
 SHNEXT:  NEXT I
 Chiudi il file
      CLOSE 1
 e assegnagli il tipo 'ARCH'
      NAME THEFILE$ AS THEFILE$,"ARCH"
RETURN
```

Le richieste di **Nuovo** vengono gestite da Donew. Dopo aver chiuso il file e la finestra precedentemente attivi, ricrea la finestra degli indirizzi e aggiorna lo stato dei menù.

```
DONEW:
  GOSUB DOCLOSE
  GOSUB WINCREATE
  MENU FILEID,3,0
  MENU FILEID,4,0
  MENU FILEID,5,0
  MENU FILEID,7,0
  MENU EDITID,0,1
  MENU EDITID,3,0
  MENU EDITID,4,0
  MENU EDITID,8,0
  MENU EDITID,9,0
  MENU SEARCHID,0,0
 RETURN
```

Apri è gestito da Doopen, che, dopo aver "azzerato" la situazione (con Donew), provvede ad emettere il dialog di richiesta del file da aprire, limitando la scelta ai files di tipo "ARCH". Una volta che il file è stato aperto, i record vengono letti, inseriti in tabella e riordinati secondo il tipo corrente.

```
DOOPEN:
  GOSUB DONEW
```

Apri il dialog di *SFGetFile* per i files di tipo "ARCH"

```
     THEFILE$=FILES$(1,"ARCH")
```

Se l'utente ha risposto con **Annulla**, termina qui

```
    IF THEFILE$="" THEN RETURN
```

Altrimenti, cambia nome alla finestra

```
     WINDOW 1,THEFILE$
```

Apri il file in input

```
    OPEN "I",1,THEFILE$
  I=0
```

leggi tutti i record

```
    WHILE NOT EOF(1)
  FOR J=1 TO 7
   INPUT#1, THETAB$(I,J)
  NEXT J
```

Inizialmente sono tutti attivi

```
    THETAB$(I,DELETED)=" "
```

e già che ci sei, riinizializza la tabella degli indici

```
    THEIND(I)=I
  I=I+1
  WEND
```

Al termine, chiudi il file

```
   CLOSE 1
```

Aggiorna il numero dei record (uno in più dell'ultimo posto occupato in tabella)

```
  NUMREC=I
```

Riordina la tabella indici

```
    GOSUB SORT
```

E sistema i menù

```
   MENU FILEID,3,1
  MENU FILEID,4,1
  MENU FILEID,5,1
  MENU FILEID,7,1
  MENU EDITID,3,1
  MENU EDITID,4,1
  MENU EDITID,8,1
  MENU EDITID,9,1
  MENU SORTID,0,1
  MENU SEARCHID,0,1
RETURN

DOPSETUP:
 RETURN
```

Non è accessibile da BASIC

Le richieste di stampa degli indirizzi vengono gestite in modo molto meno sofisticato delle corrispondenti versioni Pascal o C. I campi vengono inviati allastampante come linee di testo normale, separate da Return. La comunicazione con la stampante viene aperta con la tradizionale sintassi BASIC. L'estensione "PROMPT" indica all'interprete di emettere il dialog standard di stampa (quante copie, da che pagina, ecc.).
Vengono ovviamente stampati solo i record in quel momento attivi.

```
DOPRINT:
  OPEN "O",2,"LPT1:PROMPT"
  FOR I=0 TO NUMREC -1
   IF THETAB$(THEIND(I),DELETED)="*" THEN PRNEXT
    FOR J=1 TO 6
     PRINT#2,USING FSTR$;THETAB$(THEIND(I),J);" ";
    NEXT J
   PRINT#2,""
```

```
PRNEXT:NEXT I
CLOSE #2
RETURN
```

Del menù **Edit** si occupa Doedit, che si comporta come Dofile, richiamando le routines associate agli item selezionati.

```
DOEDIT:
ON THEITEM GOSUB UNDO, NOWHERE, MYCUT, MYCOPY, MYPASTE,
                 NOWHERE, NEWRECORD, MODIFYRECORD,
                 DELACTRECORD
RETURN

NOWHERE: RETURN

UNDO: RETURN
```

Come nelle altre due versioni, la routine di cut richiama la routine di copy e marca come cancellato il record selezionato.

```
MYCUT:
 GOSUB MYCOPY
 THETAB$(THEIND(CURRECIND),DELETED)="*"
 DIRTY=TRUE
 GOSUB SELECTLINE
 RETURN
```

La routine di copy apre il file di Clipboard e vi scrive il record corrente.

```
MYCOPY:
 OPEN "CLIP:" FOR OUTPUT AS 3
  FOR J=1 TO 6
   PRINT#3, USING FSTR$;THETAB$(THEIND(CURRECIND),J)
  NEXT J
 CLOSE 3
 RETURN
```

Più complessa la routine di paste, che richiama il sottoprogramma convfromscrap, che riceve due parametri (per var) : un flag per sapere l'esito dell'operazione e il numero dei record. Nel caso in cui lo scrap contenga informazioni compatibili, GOOD vale True e in NUMREC c'è il nuovo numero di record memorizzati; inoltre al

termine viene segnalato che l'indirizzario è stato modificato (DIRTY=True). In caso contrario, il valore di NUMREC va ripristinato a quello precedente all'ingresso nel sottoprogramma.

```
MYPASTE:
 OLDNR=NUMREC
 CALL CONVFROMSCRAP(GOOD,NUMREC)
 IF NOT GOOD THEN NUMREC=OLDNR: GOTO MYPASTERR
 IF (OLDNR>0) THEN MYPAST1
 MENU FILEID,3,1
 MENU FILEID,4,1
 MENU FILEID,5,1
 MENU FILEID,7,1
 MENU EDITID,3,1
 MENU EDITID,4,1
 MENU EDITID,8,1
 MENU EDITID,9,1
 MENU SORTID,0,1
 MENU SEARCHID,0,1
 DIRTY=TRUE

MYPAST1:
 GOSUB SORT
 RETURN
```

Nel caso in cui lo scrap sia vuoto o con dati non compatibili, mypasterr emette un alert, da cui l'utente può uscire solo premendo il mouse su OK o il tasto Return. Il meccanismo di gestione dell'alert è uguale a quello utilizzato per tutte le altre finestre.

```
MYPASTERR:
 DIALOG OFF
  WINDOW 4,,(50,50)-(400,100),-2
  PRINT "SCRAPFILE VUOTO O CON DATI NON LEGGIBILI"
  BUTTON 1,1,"OK",(200,20)-(230,35),1
  ON DIALOG GOSUB MYPASTERR1
  DIALOG ON
  ASPETTI=TRUE
  WHILE ASPETTI: WEND
  DIALOG OFF
  WINDOW CLOSE 4
  ON DIALOG GOSUB DIALOGDPTCH
  DIALOG ON
 RETURN

MYPASTERR1:
```

```
 X=DIALOG(0)
 IF X=1 OR X=6 THEN ASPETTI=FALSE
 RETURN
```

L'algoritmo di conversione da formato testo in formato indirizzo è identico a quello utilizzato nelle altre due versioni; i separatori vengono riconosciuti da *INPUT#*, e sono il Return e la virgola.

```
SUB CONVFROMSCRAP(GOOD,NUMREC) STATIC
 SHARED THETAB$(),FALSE,TRUE,DELETED,CAMPODATA
 GOOD=FALSE
 OPEN "CLIP:" FOR INPUT AS 3
 WHILE NOT EOF(3)
  FOR J=1 TO 6
   IF EOF (3) THEN GOOD=FALSE: GOTO vuoto
   INPUT#3, THETAB$(NUMREC,J)
  NEXT J
 THETAB$(NUMREC,DELETED)=" "
 THETAB$(NUMREC,CAMPODATA)=DATE$+" "+TIME$
 NUMREC=NUMREC+1
 GOOD=TRUE
 vuoto: WEND
CLOSE 3
END SUB
```

La richiesta di inserimento di un nuovo record viene soddisfatta da Newrecord, che definisce Dlogmanage come nuovo handler degli eventi di dialog, crea la nuova finestra con il richiamo di Newfilldlog; al termine riordina i dati.

```
NEWRECORD:
ON DIALOG GOSUB DLOGMANAGE
DLOGNEW=TRUE
GOSUB NEWFILLDLOG
ENDMANAGE=FALSE
DIALOG ON
WHILE NOT ENDMANAGE:WEND
DIALOG OFF
WINDOW CLOSE 2
DIRTY=TRUE
GOSUB SORT
RETURN
```

La modifica di un record segue lo stesso schema, segnalando la differenza con il flag Dlognew cui viene dato valore False.

```
MODIFYRECORD:
 IF NUMREC<=0 THEN RETURN
 ON DIALOG GOSUB DLOGMANAGE
 DLOGNEW=FALSE
 GOSUB NEWFILLDLOG
 ENDMANAGE=FALSE
 DIALOG ON
 WHILE NOT ENDMANAGE:WEND
 DIALOG OFF
 WINDOW CLOSE 2
 GOSUB SORT
 RETURN
```

La routine di gestione del dialog di inserimento/modifica è in pratica una analisi caso per caso degli eventi che si verificano. Il valore di *Dialog(0)* indica il tipo di evento. In particolare, 0 equivale a "nessun evento"; 1 rappresenta un click in un bottone attivo della finestra (gestito da Dlogbtn); 2 un click in un rettangolo di edit, il cui numero è dato da *Dialog(2)*; 5 un update (che viene ignorato); 6 un Return o Enter; 7 infine un Tab, che ha come effetto di selezionare il rettangolo di edit successivo.

```
DLOGMANAGE:
 X=DIALOG(0)
 IF X=5 OR X=0 THEN RETURN
 IF X=1 THEN DLOGBTN
 IF X=2 THEN CUREF=DIALOG(2): EDIT FIELD CUREF:RETURN
 IF X=6 THEN GOSUB DLOGOK: RETURN
 IF X=7 THEN CUREF=(CUREF MOD 6)+1:EDIT FIELD CUREF:RETURN
 RETURN


DLOGBTN:
 ON DIALOG(1) GOSUB DLOGOK, DLOGESCI, PROX, PREC, NUOVO,
                    CANCELLA, SCRIVI
 RETURN
```

Dlogok viene attivata quando l'utente conferma il record contenuto nel dialog (con un click in OK o un Return o un Enter).

DLOGOK:

 Se il record è nuovo, inseriscilo, altrimenti sovrascrivilo a quello precedente.

```
  IF DLOGNEW THEN GOSUB INSERT ELSE GOSUB WRITEREC
  DLOGNEW=FALSE
```

```
 ENDMANAGE=TRUE
RETURN
```

Insert copia il record nella prima posizione libera della tabella. Utilizza la funzione *Edit$(i)*, che ritorna la stringa di testo contenuta nell'i-esimo campo di editing della finestra attiva.

```
INSERT:
 IF NUMREC>0 THEN INSERT1
 MENU FILEID,3,1
 MENU FILEID,4,1
 MENU FILEID,5,1
 MENU FILEID,7,1
 MENU EDITID,3,1
 MENU EDITID,4,1
 MENU EDITID,8,1
 MENU EDITID,9,1
 MENU SORTID,0,1
 MENU SEARCHID,0,1
INSERT1:
 FOR I = 1 TO 6
  THETAB$(NUMREC,I)=LEFT$(EDIT$(I),20)
 NEXT I
 THETAB$(NUMREC,DELETED)=" "
 THETAB$(NUMREC,CAMPODATA)=DATE$ +" "+ TIME$
 THEIND(NUMREC)=NUMREC
 NUMREC=NUMREC+1
 RETURN
```

Writerec scrive il record nella posizione del record correntemente selezionato.

```
WRITEREC:
 FOR J=1 TO 6
  THETAB$(THEIND(CURRECIND),J)=LEFT$(EDIT$(J),20)
 NEXT J
 THETAB$(THEIND(CURRECIND),CAMPODATA)=DATE$ +" "+ TIME$
 THETAB$(THEIND(CURRECIND),DELETED)=" "
 DIRTY=TRUE
RETURN
```

L'unica funzione di Dlogesci consiste nell'assegnare il valore True al flag di terminazione dell'uso del dialog di inserimento/modifica.

```
DLOGESCI:
 ENDMANAGE=TRUE
 RETURN
```

I due bottoni Prossimo e Precedente richiamano il sottoprogramma Shiftrec (che sposta il dialog sul record adiacente) e la subroutine Oldfill, che riempie i campi di editing del dialog con il contenuto del record corrente.

```
PROX:
 DLOGNEW=FALSE
 CALL SHIFTREC(1)
 GOSUB OLDFILL
 RETURN

PREC:
 DLOGNEW=FALSE
 CALL SHIFTREC(-1)
 GOSUB OLDFILL
 RETURN
```

Shiftrec si comporta come l'omologa Pascal, aggiornando l'indice al record corrente e lo stato dei bottoni di scorrimento.

```
SUB SHIFTREC(OFFSET) STATIC
 SHARED CURRECIND,NUMREC
 CURRECIND=CURRECIND+OFFSET
 IF CURRECIND>=NUMREC-1 THEN CURRECIND=NUMREC-1: BUTTON
3,0
 IF CURRECIND<=0 THEN CURRECIND=0:BUTTON 4,0
 IF CURRECIND>0 THEN BUTTON 4,1
 IF CURRECIND<NUMREC-1 THEN BUTTON 3,1

END SUB
```

La creazione della finestra avviene per mezzo di Newfilldlog, che, aperta la finestra, cade nella routine di creazione dei bottoni e dei campi di editing. Oldfill è la parte di routine utilizzata se il campo deve essere modificato, Nfill se il campo è totalmente nuovo.

NEWFILLDLOG:

Crea una finestra, di ID 2, dal titolo "Dialog di editing", nel rettangolo..., di tipo -2 (cioè bordata e modale)

```
  WINDOW 2,"DIALOG DI EDITING",(40,40)-(440,280),-2
```

FILLDLOG:

Crea un bottone, di ID 1, di tipo 1 (pushbutton), dal titolo "OK", nel rettangolo..., attivo (1)

```
  BUTTON 1,1,"OK",(340,10)-(390,30),1
```

...
Eccetera
...
Posizionati alla riga 1, colonna 1 e scrivi il titolo del primo campo (Nome)

```
LOCATE 1,1:PRINT"Nome"
```

...
Eccetera
...
Sistema lo stato dei bottoni

```
CALL SHIFTREC(0)
```

E crea i campi di editing con il contenuto appropriato

```
IF DLOGNEW THEN NFILL
OLDFILL:
```

Crea un campo di editing, di ID 1, il cui contenuto iniziale sia il campo corrispondente del record corrente, nel rettangolo ..., di tipo 1(con cornice e senza la possibilità di inserire Return) e con giustificazione a sinistra.

```
EDIT FIELD 1,
  THETAB$(THEIND(CURRECIND),1),(90,10)-(330,30),1,1
```

.....
e così via per gli altri campi
.....
Il campo corrente è il primo...

```
CUREF=1
```

selezionalo...

```
EDIT FIELD CUREF
```

e sistema l'indicazione attivo/cancellato

```
GOSUB ADJFLAGIFDEL
RETURN
NFILL:
```

Crea un campo vuoto (vedi sopra)

```
EDIT FIELD 1,"",(90,10)-(330,30),1,1
```

.....
e così via per gli altri campi
.....
Il campo corrente è il primo...

```
CUREF=1
```

...selezionalo

```
EDIT FIELD CUREF
```

Inizialmente il record è attivo

```
LOCATE 15,1:PRINT "Attivo"
RETURN
```

Se l'utente decide di cancellare il record correntemente editato, viene eseguita Nuovo.

```
NUOVO:
IF DLOGNEW THEN GOSUB INSERT ELSE DLOGNEW=TRUE
```
Azzera i campi della finestra
```
GOTO NFILL
```

La cancellazione viene effettuata assegnando il carattere "*" al campo deleted del record. La riattivazione del record consiste nel riassegnare il carattere " " al campo.

```
CANCELLA:
 I=THEIND(CURRECIND)
 IF THETAB$(I,DELETED)="*"
         THEN THETAB$(I,DELETED)=" "
         ELSE THETAB$(I,DELETED)="*"
 GOSUB ADJFLAGIFDEL
 DIRTY=TRUE
 RETURN
```

L'indicatore di attivo/cancellato e il bottone cancella/riattiva nella finestra di dialogo viene gestito da Adjflagifdel.

```
ADJFLAGIFDEL:
 LOCATE 15,1
 IF THETAB$(THEIND(CURRECIND),DELETED)="*"
            THEN PRINT "Cancellato"
                 BUTTON 6,1,"Attiva",(310,190)-(390,210),1
                RETURN
 PRINT "Attivo"
 BUTTON 6,1,"Cancella",(310,190)-(390,210),1
 RETURN
```

Scrivi si prende carico dei click sul bottone omonimo, aggiornando la tabella se necessario, e spostandosi in avanti di un record.

```
SCRIVI:
 IF DLOGNEW THEN GOTO NUOVO
 GOSUB WRITEREC
 CALL SHIFTREC(1)
 GOSUB FILLDLOG
 RETURN
```

Il menù di **Ordinamento** viene interamente gestito dalla routine Dosort. Essa non fa nulla se l'utente seleziona il tipo già in uso; negli altri casi, sistemato il menù di ricerca (non è infatti abilitata la ricerca per data), toglie il checkmark al vecchio tipo e lo assegna al nuovo. Quindi passa ad ordinare i dati.

```
DOSORT:
 IF CURSORT = THEITEM THEN  RETURN
 IF THEITEM = SORTDATA
              THEN MENU SEARCHID,0,0
              ELSE MENU SEARCHID,0,1
 MENU SORTID,CURSORT,1
 CURSORT = THEITEM
 Metti il checkmark (2) all'elemento indicato da
Cursort
  MENU SORTID,CURSORT,2
 GOSUB SORT
RETURN
```

Sort ordina (con Shellsort), azzera l'indice corrente e stampa sulla finestra i record.

```
SORT:
 CALL SHELLSORT
 CURRECIND=0
 GOSUB  PRINTRECS
 RETURN
```

L'ultimo item del menù di **Edit** è **Cancella/Riattiva**, che modifica il campo deleted del record e aggiorna il nome dell'item.

```
DELACTRECORD:
 E' da cancellare?
  ISTODEL=THETAB$(THEIND(CURRECIND),DELETED)=" "
 IF ISTODEL THEN  THETAB$(THEIND(CURRECIND),DELETED)="*"
            ELSE  THETAB$(THEIND(CURRECIND),DELETED)=" "
 Se lo hai cancellato, nel menù deve comparire Riattiva, altrimenti
 Cancella.
 IF ISTODEL
    THEN MENU EDITID,9,1,"Riattiva                  "+CMD$+"D"
    ELSE MENU EDITID,9,1,"Cancella                  "+CMD$+"D"
 GOSUB SELECTLINE
 RETURN
```

L'ultimo menù è quello di **Ricerca**, che viene gestito con Dosearch. Questa routine fa uso di un dialog modale di tipo 3 (con bordo semplice) e tratta gli eventi relativi con Searchmanage. Poichè durante l'esecuzione di un handler gli eventi sono disabilitati, nel caso di fallimento di una ricerca l'alert non potrebbe più venir rilasciato (poichè i click al suo interno non verrebbero "sentiti"). Per evitare ciò abbiamo utilizzato un ciclo di ricerca con due flag: uno che indica se la fase di ricerca è terminata perchè l'utente ha clickato su "Esci" (Searchend), l'altro se è terminata una ricerca(Loop). Se l'esito della ricerca è negativo occorre emettere un alert, attendere il suo rilascio e quindi tornare ad aspettare eventi sul dialog di ricerca; altrimenti si aggiorna l'indice del record corrente, si riscrive la finestra degli indirizzi e infine si riapre il dialog di ricerca.

```
DOSEARCH:
 ON DIALOG GOSUB SEARCHMANAGE
 WINDOW 2,"RICERCA",(100,200)-(310,310),-3
 PRINT "Cerca il record che..."
 EDIT FIELD 1,SEARCH$,(10,80)-(200,100),1,1
 BUTTON 1,1,"cerca",(160,20)-(200,40),1
 BUTTON 2,1,"esci",(160,50)-(200,70),1
 BUTTON 3,1,"contiene",(10,50)-(140,62),3
 BUTTON 4,2,"inizia con",(10,65)-(140,77),3
 ISINIT=TRUE
 SEARCHEND=FALSE
 LOOP=TRUE
 DIALOG ON
 GOON:
   WHILE LOOP
   WEND
 IF SEARCHEND THEN FINESEARCH
 LOOP=TRUE
 IF TROVATO THEN GOON
 DIALOG OFF
 WINDOW 3,,(50,50)-(300,110),-2
  LOCATE 1,1
  PRINT "Record non trovato":BEEP
  BUTTON 1,1,"OK",(210,20)-(240,38),1
  ASPETTI=TRUE
  ON DIALOG GOSUB NONTROVATO:DIALOG ON
  WHILE  ASPETTI :WEND
  ON DIALOG GOSUB SEARCHMANAGE
  WINDOW CLOSE 3
  GOSUB REFRESH
  GOTO GOON
```

L'handler dell'alert "Non trovato" aspetta solo che l'utente prema il bottone del mouse su OK oppure batta Return o Enter.

```
NONTROVATO:
  X=DIALOG(0)
  IF X=1 THEN ASPETTI=FALSE: RETURN
  IF X=6 THEN ASPETTI=FALSE
  RETURN
```

Al termine della fase di ricerca occorre ripristinare l'handler relativo alla finestra degli indirizzi e chiudere la finestra di ricerca.

```
FINESEARCH:
   ON DIALOG GOSUB DIALOGDPTCH
   WINDOW CLOSE 2
   RETURN
```

La routine di handling della finestra di ricerca si occupa solo degli eventi di tipo 1 (bottoni) e 6 (Return o Enter).

```
 SEARCHMANAGE:
   X=DIALOG(0)
   IF X=0 THEN RETURN
   IF X=6 THEN GOSUB CERCA:RETURN
   IF X=1 THEN  GOSUB BTNPRESS:  RETURN
   RETURN
```

A seconda del bottone premuto, Btnpress cede il controllo alla routine associata.

```
BTNPRESS:
  WHICHBTN=DIALOG(1)
  ON WHICHBTN GOSUB CERCA,ESCI,CONTIENE,INIZIA
RETURN
```

Cerca esegue la ricerca della stringa contenuta nell'Edit field 1 del dialog.

```
 CERCA:
```

Loop viene messo a False, per indicare che una ricerca è stata effettuata

```
  LOOP=FALSE
```

```
 SEARCH$= EDIT$(1)
 GOSUB FIND
 TROVATO=IND>=0
 IF NOT TROVATO THEN RETURN
```
Salva la finestra corrente
```
     SW=WINDOW(0)
```
Ridirigi l'output sulla finestra degli indirizzi
```
     WINDOW OUTPUT 1
```
Deseleziona la linea corrente, che è sicuramente visibile
```
     GOSUB UNSELECTLINE
```
Aggiorna il nuovo indice corrente...
```
     CURRECIND=IND
```
... e il contatore di scroll
```
     IF IND<PAGEREC THEN SC=0 ELSE SC=IND-PAGEREC+1
```
Se cambia il valore di scroll occorre riscrivere gli indirizzi, altrimenti basta selezionare la linea opportuna
```
  IF SC<>SCROLLCOUNT THEN SCROLLCOUNT=SC: GOSUB PRINTRECS
                          ELSE GOSUB SELECTLINE
```
infine ridefinisci la finestra attiva
```
     WINDOW SW
 RETURN
```

Find effettua la ricerca della stringa search$ all'inizio o all'interno del campo cursort dei record, a seconda del valore del flag isinit. L'algoritmo è lo stesso usato nella versione Pascal, cui si può far riferimento per ulteriori chiarimenti.

```
FIND:
 TOBEFOUND=TRUE
 CONTINUE=TRUE
```
Parti dal record successivo a quello corrente
```
     I= CURRECIND+1
 IF I>=NUMREC THEN I=0
 WHILE TOBEFOUND AND CONTINUE
   K=THEIND(I)
   A$=THETAB$(K,CURSORT)
```
Usa la funzione BASIC instr, che ritorna la posizione di inizio di una sottostringa in una stringa (0 se non è contenuta)
```
   J=INSTR(A$,SEARCH$)
```
Se il risultato è 1 (search$ è all'inizio) oppure è contenuta (j>0) e il tipo di ricerca è "contiene...", abbiamo trovato il record.
```
          IF J=1 OR (J>0 AND NOT ISINIT)
                     THEN TOBEFOUND=FALSE :GOTO NEXTSTEP
```

Se siamo tornati al punto di partenza del ciclo occorre interrompere

```
    IF I=CURRECIND THEN CONTINUE=FALSE
    I=I+1
    IF I>=NUMREC THEN I=0
 NEXTSTEP:WEND
```

Assegna il valore alla variabile di risposta

```
      IF TOBEFOUND THEN IND=-1 ELSE IND=I
RETURN

 ESCI:
  LOOP=FALSE
  SEARCHEND=TRUE
  RETURN
```

Le due routines realtive ai radiobuttons "inizia con..." e "contiene..." modificano il valore del flag isinit e lo stato dei due bottoni.

```
 CONTIENE:
  ISINIT=FALSE
```

Seleziona il bottone 3 ("contiene...")

```
      BUTTON 3,2
```

e deseleziona il bottone 4 ("inizia con...")

```
      BUTTON 4,1
  RETURN

 INIZIA:
  ISINIT=TRUE
  BUTTON 3,1
  BUTTON 4,2
  RETURN
```

Il sistema di sviluppo Aztec C

Aztec C è un sistema di sviluppo per il Macintosh basato su di un compilatore del linguaggio "C" ed un interprete di comandi line-oriented simile, almeno come filosofia, alla shell del sistema operativo Unix. L'interazione col sistema è basata su comandi inviati esclusivamente da tastiera: il mouse rimane praticamente inutilizzato. Attraverso i comandi dati alla shell è possibile gestire qualunque operazione effettuabile con il Finder, con in più una discreta quantità di tools utilissimi se non indispensabili per lo sviluppo di software. Esistono anche un certo numero di comandi di estrema utilità per la gestione di Mac in generale, quali comandi per copiare oggetti e/o interi dischi, per rimuoverli rapidamente o per ridenominarli.

La shell fornisce una visione apparentemente gerarchica del file-system simulando il concetto di directory mediante l'uso di / per distinguere i diversi livelli di profondità di un oggetto. (Esistono dei limiti oggettivi alla profondità reale della struttura gerarchica a causa della limitazione sulla lunghezza dei nomi dei files imposta dal sistema operativo di Macintosh)

Una delle caratteristiche più interessanti, agli effetti dello sviluppo di applicazioni anche complesse, è la presenza dei tools di cui sopra: particolarmente interessanti sono *Make* (un sistema per il mantenimento e l'aggiornamento di applicazioni complesse composte da diversi moduli), programmi per verificare le differenze tra più files, altri per la ricerca delle occorrenze di patterns in più files, gestori di librerie e di archivi e molti altri.

Esiste infine *Z,* un editor screen-oriented simile, come filosofia e sintassi dei comandi, a *vi* , editor tipico del sistema Unix. I files prodotti da Z sono di tipo TEXT e possono essere quindi caricati anche da altri editors quali Edit o MacWrite. (Insieme a Z viene comunque distribuito anche Edit, per chi fosse irrimediabilmente assuefatto al ...mouse!)

Il Linguaggio

Per quanto riguarda la programmazione vera e propria, il sistema Aztec C comprende un compilatore C standard (o quasi), un assembler per il 68000, un linkage editor e due Resource compilers: uno, chiamato *RMaker*, è quello standard Apple (simile a quello del Lisa-Workshop), l'altro, chiamato *RGen*, è orientato all'uso con la shell non essendo interattivo ed è compatibile con RMaker a livello di sorgente.

Compilatore (*cc*)

Il compilatore Aztec C riconosce sorgente C standard con alcune restrizioni rispetto ai compilatori Unix (dalla V7 in poi) e Xenix: le più significative sono l'assenza dei tipi enumerativi, l'assenza dei Bit-Fields e l'impossibilità di passare struct's come argomento a funzioni. Il compilatore genera sorgente per l'assembler Aztec e attiva automaticamente l'assembler se non diversamente specificato. A differenza dei compilatori C Unix questo non può essere usato in modo *load-and-go*: E' cioè necessario attivare esplicitamente il linker-loader e citare tutte le librerie anche per produrre un binario eseguibile composto da un solo modulo.

Assembler (*as*)

L'assembler Aztec riconosce sorgente per il processore MC68000 compatibile con il set di istruzioni e relativi mnemonici descritti nel *MC68000 User's Manual* e genera un modulo binario rilocabile che deve in ogni caso essere caricato dal linker-loader. Sono supportate un discreto numero di direttive di controllo dell'assemblaggio, di macro-definizioni e di controllo del listing.

Linkage-Editor (*ln*)

Il linker Aztec produce un file binario eseguibile linkando moduli binari rilocabili prodotti dall'assembler *as* e librerie di moduli

rilocabili.
I binari prodotti possono essere di tre tipi principali:

- Applicazioni, attivabili dal Finder o dalla Shell o da altre applicazioni.
- Device drivers.
- Desk-Accessories.

Il linker-loader permette di suddividere il codice di un'applicazione in segmenti, gestiti a run-time dal *segment-loader* di Mac.
Il linker scandisce le librerie e i moduli oggetto in una sola passata, risolvendo i riferimenti in avanti: l'ordine di linkaggio è perciò significativo.

Resource Compilers (*RMaker, RGen*)

Per compilare le risorse di un'applicazione o, in generale, per produrre dei files di risorse sono disponibili due resource compilers.

RMaker: è il resource compiler dell'Apple MDS. È molto simile a quello del Lisa-Workshop descritto in *Inside Macintosh*, a parte alcune differenze di sintassi.

RGen: è fornito insieme al sistema di sviluppo Aztec C, non è interattivo ed è compatibile con il formato sorgente di RMaker. Alcuni tipi di risorse non sono supportati (p. es. CODE, DRVR, FONT). Differisce da RMaker per l'estensione dei nomi di files.

Uso del sistema di sviluppo.

Tutte le operazioni di sviluppo software con il sistema Aztec C devono essere effettuate usando come command processor la shell Aztec, questo perchè quasi tutti i tools di sviluppo, a partire dal compilatore, sono programmi che possono essere lanciati esclusivamente dalla shell. È possibile utilizzare diversi dischetti, anche più di quanti sono i drive disponibili: la shell (come del resto il Finder) tiene traccia di tutti i dischetti *montati* e all'occorrenza richiede di inserire il dischetto opportuno se questo non è già in qualche drive. E' inoltre possibile espellere direttamente un disco con una semplice combinazione di tasti.

L'ambiente operativo è, come già accennato, simile alla shell del sistema operativo Unix; anche qui è significativo il concetto di directory, di pathname e di directory corrente. (È da tener presente che il concetto di directory non ha nulla a che vedere con quello di *Folder* Macintosh)
Comandi quali ls, mv, cp permettono di listare il contenuto della directory corrente, spostare o copiare files da una directory all'altra o cambiarne il nome etc.

I passi principali dello sviluppo di un'applicazione Macintosh con Aztec C possono essere riassunti in:

- Editing dei vari moduli sorgenti mediante *Z* oppure *Edit*.

- Creazione del *Makefile*, il file usato da *Make*, in cui sono espresse le dipendenze tra i vari moduli e il modo per aggiornarli.

- Esecuzione di Make che automaticamente compila i moduli modificati nel periodo di tempo trascorso dall'ultima compilazione, li linka insieme con le librerie opportune (create magari dal programmatore stesso) e attiva il resource compiler per produrre l'applicazione finale con le proprie risorse.

Per la scrittura e la manutenzione dei moduli sorgenti sono di estrema utilità tools come *grep*, che permette di ricercare le occorrenze di patterns (espressi con una sofisticata meta-notazione) all'interno di diversi files, *diff* che riporta le

differenze tra due files, e *ctags* che, usato insieme a particolari comandi dell'editor *Z,* permette di passare direttamente all'editing di un file in cui è definitita una routine che si vuole modificare.

Uso del linguaggio

In un'applicazione Macintosh giocano un ruolo fondamentale le routines del Toolbox, cioè quel gruppo di funzioni predefinite (residenti per lo più in ROM) che consentono di dare ad un'applicazione Mac le caratteristiche tipiche di questo quali finestre, menù, mouse etc.

Come è noto queste routines sono progettate per essere chiamate da un programma Pascal e quindi per un compilatore C che produce codice che fa uso del Toolbox esiste il problema di fornire la possibilità di accedere a tali routines da un linguaggio che, per quanto riguarda le convenzioni di passaggio dei parametri e l'uso dei registri, differisce notevolmente dal Pascal.

Gli approcci possibili per la soluzione di tale problema sono essenzialmente due: le cosiddette *Glue-routines* e la traduzione diretta da parte del compilatore.

L'uso delle Glue-routines consiste nel sostituire, a livello di programma, le vere chiamate al Toolbox con delle funzioni, con interfaccia C, che a loro volta effettuano le opportune conversioni dei dati e chiamano in modo opportuno le vere routines del Toolbox.

L'altro approccio consiste nel far sì che il compilatore produca codice diverso quando incontra una chiamata ad una routine del Toolbox. Questa seconda soluzione è più efficiente dal punto di vista del tempo di esecuzione e della compattezza del codice, ma per alcuni aspetti risulta meno flessibile per il programmatore. L'approccio seguito dal compilatore Aztec è il secondo: esiste infatti la direttiva (in effetti è qualcosa di simile ad una storage-class) *pascal* che indica al compilatore che la routine che segue deve essere compilata o chiamata con le convenzioni di passaggio dei parametri usate dal Pascal. Le routines del Toolbox sono perciò dichiarate tutte con questa direttiva e devono essere

usate con alcuni accorgimenti dovuti essenzialmente ai seguenti problemi:

i) Diverso formato delle stringhe in pascal e C.
ii) Diverso formato dei valori booleani.
iii) Passaggio delle struct's *by reference*.
iv) Passaggio di puntatori a funzioni.

i) Esiste il tipo *String(x)* (è una macro) che definisce una stringa tipo Pascal nel seguente modo:

```
struct {
    unsigned char  lunghezza;
        char  testo[x]; }
```

Ad esempio il tipo *Str255*, tipico del Lisa Pascal, è definito come String(255).
Se si passa ad una routine del Toolbox una stringa di questo tipo è allora necessario passarne l'indirizzo (&) trattandosi di una struct, se invece si vuole passare una stringa C (puntatore a char) è necessario convertirne opportunamente il formato; per fare ciò sono a disposizione due routines di libreria *ctop* e *ptoc* che convertono il formato della stringa passata rispettivamente in Pascal e C ritornando il puntatore alla stringa stessa. (È da tener presente che l'operazione di conversione è distruttiva, cioè opera effettivamente sulla stringa passata).

Infine per le costanti di tipo stringa Pascal esiste la notazione **"\P*<testo>*"** che corrisponde ad una stringa in formato Pascal con testo *<testo>*: costanti di questo genere possono essere direttamente passate a routines del Toolbox che aspettano stringhe Pascal come argomento. - p.es. DrawStr("\PStringa Pascal"); -

ii) In Pascal i booleani vengono allocati in due bytes con il bit significativo nel byte alto (quindi TRUE è un valore ≥ di 256, FALSE il contrario), in C i valori di espressioni booleane sono di tipo int con la convenzione 0 = FALSE, altro = TRUE . Quindi non bisogna passare direttamente al Toolbox espressioni booleane, ma solo le costanti TRUE o FALSE che sono definite opportunamente.

iii) Dato che in Pascal il passaggio di parametri dichiarati *var* corrisponde a passare l'indirizzo della struttura stessa, è necessario passare, a routines del Toolbox che aspettano argomenti *var*, l'indirizzo del parametro stesso.

Inoltre dato che nella implementazione Aztec del C non è consentito il passaggio di struct's, è necessario passarne il puntatore anche a routines che aspettano argomenti *by value*. Tuttavia il problema non è così semplice: quanto detto nell'ultimo caso è vero, sempre che la *size* della struct che si dovrebbe passsare sia maggiore di quattro (4) bytes! In caso contrario non si deve passare il puntatore bensì la struct stessa effettuando però un *cast* a *long*. Per fare ciò è fornita una utile macro di nome *pass()* che effettua direttamente la conversione. Per esempio:

```
EventRecord  myevent;

FindWindow(pass(myevent.where));
```

Attenzione però: ciò deve essere fatto solo nel caso che il parametro atteso dalla routine sia *by value*: Altrimenti usando *pass()* la struct viene passata in realtà come long e quindi ricopiata sullo stack, mentre la routine si aspetta l'indirizzo della variabile da modificare, e dunque la copia reale (il parametro attuale) non viene affatto modificato. Quindi esemplificando:

```
Point   mypt;

SetPt(pass(mypt));        Errato!
SetPt(&mypt);             Corretto!
```

Comunque l'unico tipo standard che verosimilmente capita di passare a routines del Toolbox tramite *pass()* è Point: la fatica è quindi minima.

iv) Esistono alcune routines del Toolbox che aspettano come argomenti dei puntatori a funzioni (ProcPtr). Quando si deve passare una funzione scritta in C ad una routine del Toolbox, per esempio *TrackControl*, è necessario, per quanto detto prima riguardo al passaggio dei parametri tra funzioni C e Pascal, che questa sia dichiarata come *pascal*, in modo che, quando viene eseguita dalla routine del Toolbox cui è stata passata, funzioni

correttamente (cioè in maniera Pascal).

Un'ultima annotazione interessante riguardo al compilatore riguarda l'uso dei long e int per esprimere una *size* (tipicamente per passarla a routines del memory manager).
In realtà tutte le sizes devono essere espresse come long, tuttavia capita spesso di usare l'espressione *sizeof* come argomento a routines come p. es. NewHandle(): attenzione!
Il tipo di una espressione che coinvolge *sizeof* è *unsigned int* e non long. Inoltre, dato che i puntatori sono entità a 32 bits, l'aritmetica tra puntatori deve essere effettuata con costanti ed espressioni di tipo long per essere al sicuro da errori difficilmente riconoscibili. Infine va sottolineata la necessità di lockare gli handles prima di dereferenziarli! (Ad esempio per fare una ricerca su un array mediante il puntatore e non l'handle).

Come ultimi commenti sono interessanti alcune features utili del compilatore, quali la generazione del sorgente assembler con gli statements C inclusi come commento o la disponibilità di simboli predefiniti come il nome della funzione corrente etc., e del linker, come la possibiltà di generare una symbol table con offsets rilocabili che può essere un utilissimo supporto ai rudimentali debuggers disponibili sul Mac. È invece negativo il fatto che esista un limite molto stretto al livello di nesting di *#include* (3) per il preprocessor: Questo crea alcuni problemi quando si includono headers che a loro volta ne includono altri; dato che ciò accade per quasi tutti gli headers di sistema, è necessario seguire un opportuno ordine negli statements di include.